Java

Java

- Java is object oriented.

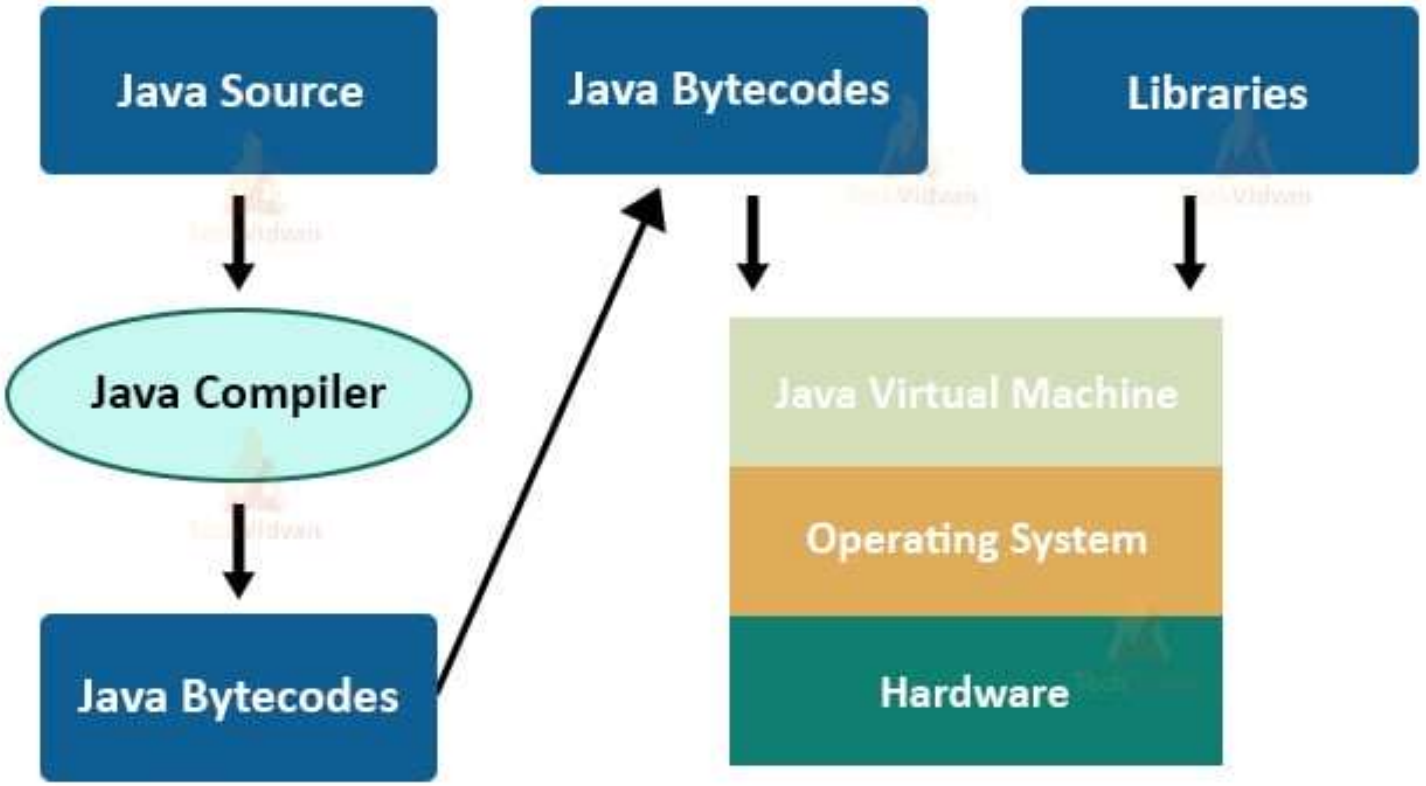
- Java works on most platforms.
- While C/C++ programs are platform specific.

- Java is network enabled. It is trivially simple to write code in Java that works across networks.

What is JVM?

- It is a hypothetical processor which executes java programs.
- The java compilers produces binary byte code designed to execute on JVM rather than on a PC or Sun Workstation.
- It is abbreviated as Java Virtual Machine.

Working of JVM



Declaring Variables

- Java requires that you ***declare*** every variable before you can use it.

- You can declare the variables in several ways. Often, you declare several at the same time.

```
int y, m, x //all at once
```

or one at a time.

```
int y; //one at a time
int m;
int x;
```

Declaring Variables (cont…)

- You can also declare the variables as you use them.

```
int x = 4;
int m = 8;
int b = -2;
```

- However, you MUST declare variables by the time you first refer to them.

Constants

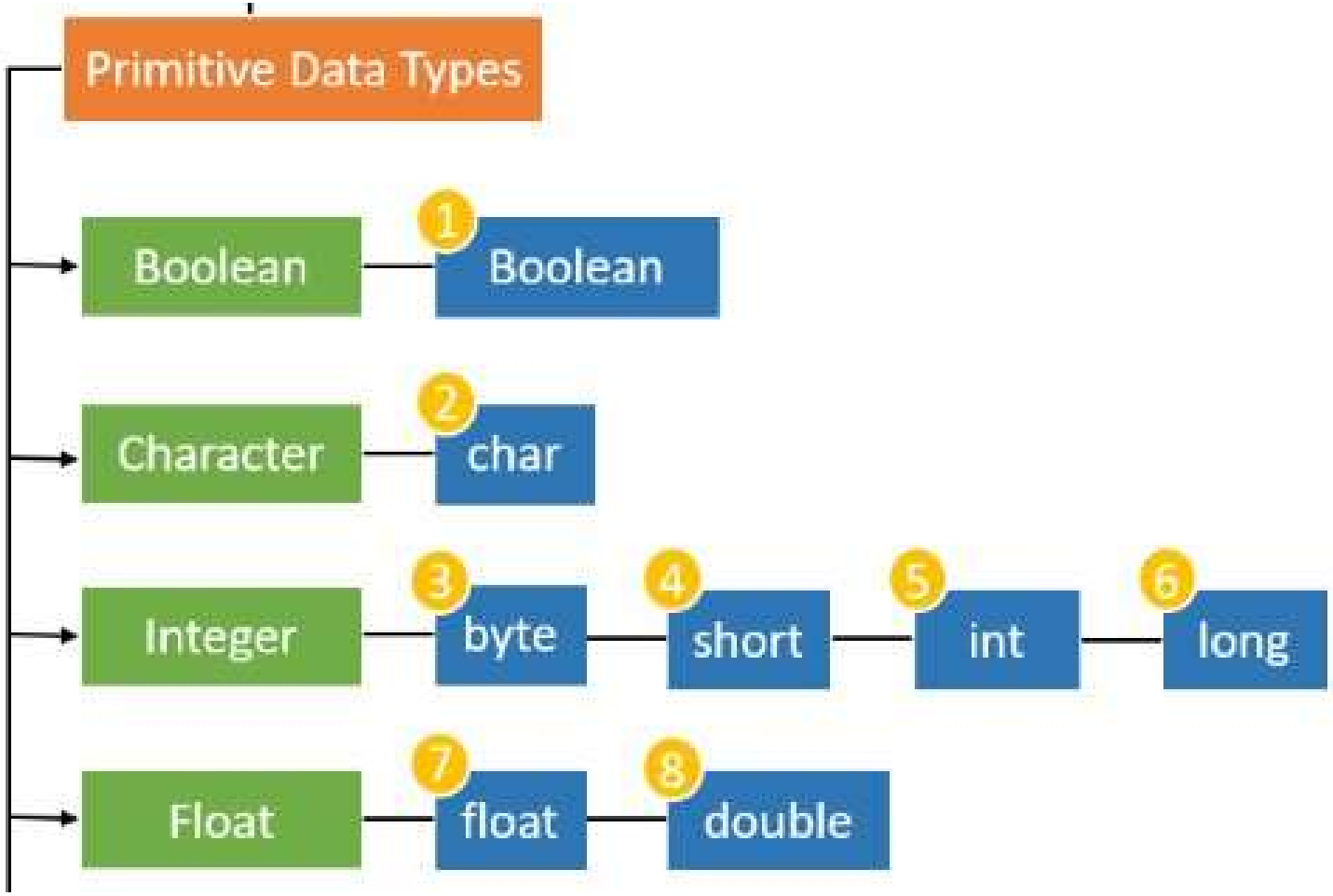
- Constants can be defined by using the final modifier.
- It is a good practice to CAPITALIZE symbols referring to constants.
- Examples

```
final float PI = 3.1416;
final int NUMBER_OF_DAYS_IN_MONTH = 30;
```

Data Types

Type	**Contents**
boolean	true or false
byte	Signed 8-bit value
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit floating point
double	64-bit floating point
char	16-bit unicode character

Data Types


Primitive Data Types
Boolean
1
Boolean
Character
2
char
Integer
3
byte
4
short
5
int
6
long
Float
7
float
8
double

Data Type Conversions

- Any "wider" data type can have a lower data type assigned directly to it and the promotion to the new type will occur automatically.

- For example, If *y* is of type **float** and *j* is of type **int** then you can write:
  - `float y; //y is of type float`
  - `int j; //j is of type int`
  - `y = j; //convert int to float`

  to promote an integer to a float.

Data Type Conversions (cont…)

- You can reduce a wider type to a narrower type using by *casting i*t. You do this by putting the data type name in parentheses and putting this name in front of the value you wish to convert:

- For example,

```
j = (int)y; //convert float to integer
```

Boolean Data Type

- Boolean variables can only take on the values represented by the reserved words **true** and **false**.

Unlike C, you cannot assign numeric values to a boolean variable and you cannot convert between **boolean** and any other type.

Numeric Data Types

- Any number you type into your program is automatically of type **int** if it has no fractional part or of type **double** if it does.

- For example in a program if a number, say 5.5 is used then it will be of double data type by default.

Numeric Data Types (cont…)

- If you want to indicate that it is a different type, you can use various suffix and prefix characters to indicate what you had in mind.

- For example,

```
float loan = 1.23f; //float
long pig = 45L; //long
long color = 0x12345; //hexadecimal
```

Simple Java Program

```
import java.io.*
public class Add2 {

 public static void main(String argv[ ]) {
     double a, b, c;
      a = 1.75;
      b = 3.46;
      c = a + b;

      // print out sum
      System.out.println("sum = " + c);
   }
 }
```

Output : sum = 5.21

import statement

- You must use the **import** statement to define libraries or *packages* of Java code that you want to use in your program
- This is similar to the C and C++ **#include** directive.

“main” Method

- The program starts from a function called **main** and it must have *exactly* the form shown here:

```
public static void main(String argv[])
```

or

```
public static void main(String[] argv)
```

Comments

- Single line comments start with "//" (double forward slash)

- Multiple lines can be commented by enclosing the required code block between /* and */

Class Definition

- Every program module must contain one or more classes.
- The class and each function within the class is surrounded by *braces* ({ }).
- Like most other languages the equals sign is used to represent assignment of data.

String concatenation

- You can use the “+” sign to combine two strings. The string “sum =” is concatenated with the string representation of the double precision variable **c**.

Arithmetic Operators

+	Addition
-	Subtraction, Unary Minus
*	Multiplication
/	Division
%	Modulus

Assignment Operators

=	Assignment
+= -= *= /= **etc**	Compound assignment

Increment & Decrement Operators

- Java allows you to express incrementing and decrementing of integer variables using the “++” and “--” operators.

- For example

```
i = 5; j = 10;
x= i++  //x = 5, then i = 6
y = --j; //y = 9 , j = 9;
```

Control structures

Making decisions in Java

- The familiar if-then-else of Visual Basic has its analog in Java. Note that in Java, however, we do not use the “then” keyword. For Example,

```
if ( y > 0 )
  z = x / y;
```

- Parentheses around the condition are REQUIRED in Java.

Making decisions in java (cont…)

- If you want to have several statements as part of the condition, you must enclose them in braces:

```
if (y > 0) {
 z = x / y;
 System.out.println("z = " + z);
}
```

Comparison Operators

Java	**Meaning**
>	Greater than
<	Less than
==	Is equal to
!=	Is not equal to
>=	Greater than or equal to
<=	Less than or equal to
!	Not
Note :- All of these operators return boolean results	

Logical operators

Operator	Meaning
&	Logical AND
|	Logical OR
&&	Shortcut Logical AND
||	Shortcut Logical OR

switch statement

```
switch(expression) {
 case constant : statements
 case constant : statements
 ----
 ----
 default : statements
}
```

switch statement - example

```
switch(number) {
 case 1: System.out.println("One");
         break;
 case 2: System.out.println("Two");
         break;
 case 3: System.out.println("Three");
         break;
 default : System.out.println("Invalid");
}
```

Loops

```
while(boolean-expression)
        statement
```

```
do
    statement
        while(boolean-expression);
```

```
for(initialization; boolean-expression; step)
        statement
```

break and continue

- You can also control the flow of the loop inside the body of any of the iteration statements by using **break** and **continue**
- **break** quits the loop without executing the rest of the statements in the loop
- **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration
- For nested loops, labels can be used along with these statements to specify the loop

Classes and Objects

Class

- It is a basic unit in java programming language.
- It provides a structure for objects.
- Contract – A combination of methods, data and semantics.

Java Class Structure

```
package <package_name>;

import <other_packages>;

public class ClassName {
    <variables(also known as fields)>;

    <constructor(s)>;

    <other methods>;
}
```

Simple class

```
class BankAccount {
  private double balance = 0.0;
}
```

Class Members

1. Fields
2. Methods
3. Classes – Nested classes
4. Interfaces – Nested interfaces

Fields

- Class variables are called fields.
- Fields are data variables associated with a class and its objects.
- Instance variables – associated with objects
- Static variables – associated with class

Field Initialization

- When a field is declared it can be initialized by assigning it a valued of the corresponding type
  - `double zero = 0.0;` // constant
  - `double sum = 4.5 + 3.7;` // constant expression
  - `double zeroCopy = zero;` // field
  - `double rootTwo = Math.sqrt(2);` // method invocation
  - `double someVal = sum + 2 * Math.sqrt(rootTwo)`

Field Initialization *contd..*

- If a field is not initialized a default initial value is assigned to it depending on its type.

Type	**Default value**
boolean	false
char	‘\u0000’
int types	0
float	0.0f
double	0.0
Object ref	null

Final fields

- A final variable is one whose value cannot be changed after it has been initialized.
- Ex:

```
final double PI = 3.141592;
```

Methods

- Methods also are members of a class
- Method should have a type
- Type of a method is the type of data it returns
- If the method does not return a value its type is void
- ‘this’ can be used to refer instance variable explicitly

Method Overloading

- **Rules for method overloading：**

1. Overloaded methods *must* change the argument list.
2. Overloaded methods can change the return type.
3. Overloaded methods can change the access modifier.

Method Overloading

```
public static void main(String[] args) {
   Sales s = new Sales();

   System.out.println(s.computeSales(100));
   System.out.println(s.computeSales(100,2));
   System.out.println(s.computeSales(100,2,30));
}

class Sales {
   double computeSales(double price) {
         double sales;
         sales = price;
         return sales;
   }
   double computeSales(double price, int qty) {
         double sales;
         sales = price * qty;
         return sales;
   }
   double computeSales(double price, int qty, double discount) {
         double sales;
         sales = (price * qty) - discount;
         return sales;
   }
}
```

```
100.0
200.0
170.0
```

Static variables

- Variables shared by all objects of a class are called static fields or class variables.

```
static int nextID;
```

- Used to declare common variables for all the objects
- When accessed externally it must be accessed via class name or any object reference

  Ex: System.out

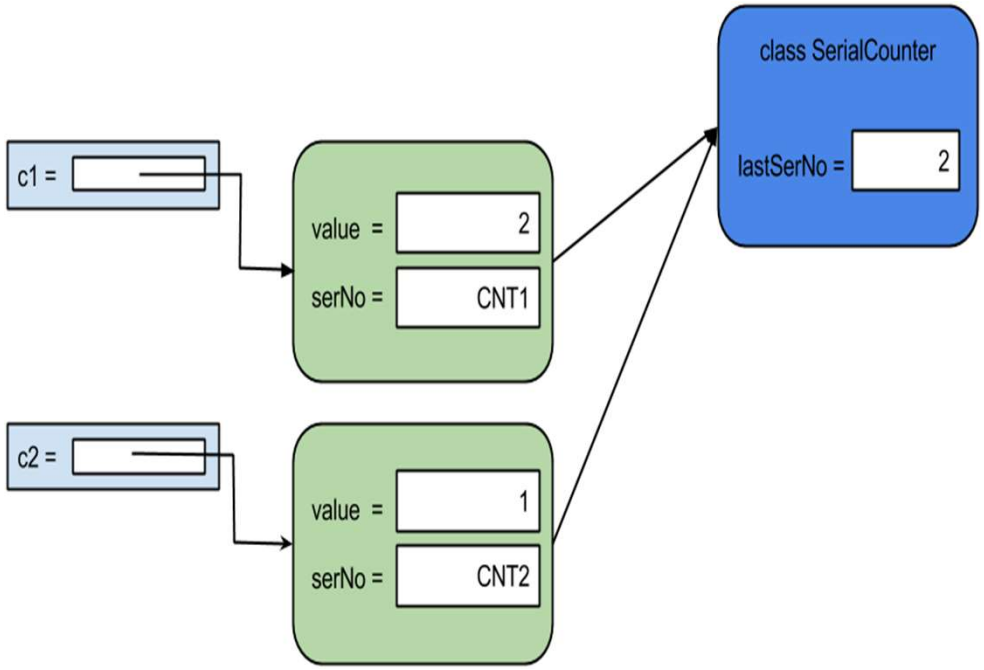
instance vs static

```
public class SerialCounter extends Counter {
    private String serNo;

    private static int lastSerNo = 0;

    public SerialCounter() {
        lastSerNo = lastSerNo + 1;
        serNo = "CNT" + lastSerNo;
    }

    public String getSerialNumber() {
        return serNo;
    }
}
```



Static methods

- static methods are class level methods
- They can be called using objects reference and class name
- They cannot access instance data
- They cannot call instance methods
- They can only call other static methods

Access control

- It provides a way to who has access to what members of a class.
  - **`private`** – accessible only in the class itself.
  - **`Default`** – accessible in
    - classes in the same package
    - class itself
  - **`protected`** – accessible in
    - subclasses of any package
    - classes in same package,
    - the class itself.
  - **`public`** – accessible
    - anywhere the class is accessible.

Creating Objects

- Object of a class is created using the keyword – new
- Ex:-

```
BankAccount anAccount = new BankAccount()
anAccount.balance = 1000.00;
```

- You never delete objects. JVM manages memory for you using *garbage collection*.

Constructors

- Constructors are blocks of statements that can be used to initialize an object.
- Constructors have the same name as the class they initialize.
- Constructors take zero or more arguments.

Constructors (contd)

EX:

```
class BankAccount {
      double balance = 0.0
      BankAccount(double initialBalance) {
            balance = initialBalance
      }
  }
```

Using constructor to create objects:

```
BankAccount anAccount = new BankAccount(1000.00);
```

Constructors (contd)

- All Java classes have constructors that are used to initialize a new object of that type
- A constructor has the **SAME NAME** as the class
- A constructor **DOESNOT** have return type.
- For example, a no argument constructor for *Stack* class can be

```
public Stack() {
    items = new Vector(10);
}
```

Constructors -Overloading

- Java supports name overloading for constructors so that a class can have any number of constructors, all of which have the same name

- For example, constructors for the stack classes can be
  - `public Stack()` // no argument ctor
  - `public Stack(int initialSize)` // 1 argument ctor

Default Constructor

- When writing your own class, you **DON'T HAVE** to provide constructors for it
- The default no argument constructor is automatically provided by the compiler for any class that contains no constructors
- If a constructor with arguments is provided, default constructor is not automatically created by the compiler

Create Object

- A class provides the blueprint for objects
- Variable of class type is object reference
- Unless assigned with object reference, variable value is null

```
Point p1 = new Point(23, 94);
Rectangle r1 = new Rectangle(origin_one, 100, 200);
Point p2;    // value of p2 is  null
P1 = new Point(23,34);  // now p1 refers a new object
```

- **Referencing an Object's Variables (instance variables)**

  objectReference.variableName
- **Calling an Object's Methods (instance methods)**

  objectReference.methodName(argumentList);

Life Cycle Of an Object

Cleaning Up Unused Objects

- The Java runtime environment deletes objects when it determines that they are no longer used.
- This process is called *garbage collection*.
- An object is eligible for garbage collection when there are no more references to that object.
- We can explicitly drop an object reference by setting the variable to the special value *null*

Garbage Collector

- JRE has a garbage collector that periodically frees the memory used by objects that are no longer referenced.

Encapsulation

- Encapsulation is one of the four fundamental object-oriented programming concepts.
- The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it.
- Encapsulation covers, or wraps, the internal workings of a Java object.
  - Data variables, or fields are hidden from the user of the object.
  - Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
  - As long as the services do not change, the implementation can be modified without impacting the user.

Public and Private Access Modifiers

- The *public* keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The *private* keyword, applied to fields and methods, allows access only to other methods within the class itself.

- One way to hide implementation details is to declare all of the fields *private and methods as public*

String objects

String

- The String class represents character strings
- All string literals in Java programs, such as "abc", are implemented as instances of this class.
- Strings are constant; their values cannot be changed after they are created
- The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase
- The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

String (contd)

- Ways of creating String objects :

```
String s = “Hello”
String s = new String(“hello”);

char[ ]  ch = { ‘a’,’b’,’c’};
 String s = new String(ch);
```

String methods

Some of the methods

```
char charAt(int index)
String concat(String)
boolean equals(Object)
boolean equalsIgnoreCase(String)
int indexOf(String str)
int length()
String replace(char old, char new)
String substring(int begin, int end)   // begin to end – 1
String toLowerCase()
String toUpperCase()
String trim()
static String valueOf(alltypes)
```

StringBuffer & StringBuilder

Both classes represent mutable string objects

Both have same methods

StringBuffer is threadsafe, StringBuilder is not

Constructors

```
StringBuilder( )    // initial capacity 16
StringBuilder(int capacity)
StringBuilder(String st)      //create with capacity 16 + length of st
```

StringBuffer & StringBuilder

Some methods

StringBuilder append(alltypes)
int capacity( )
char charAt(int index)
int capacity()
StringBuilder delete(int start, int end) //start to end – 1
StringBuilder deleteCharAt(int index)
int indexOf(String)
StringBuilder insert(int offset, alltypes)
int length( )
StringBuilder replace(int start, int end, String new)
StringBuffer reverse( )

Arrays

- Arrays provide ordered collections of elements.
- Components of array can be primitive types or references to objects, including references to other arrays.
- Arrays themselves are objects and extend the class Object
- Examples:

```
int[] x = new int[3];
int y[] = new int[3];
```

Arrays (cont…)

- An `ArrayIndexOutOfBoundsException` is thrown if the index is out of bounds.
- The index expression must be of type int
- Implicit length variable used to know the size of the array
- An array with length zero is said to be an empty array.

Arrays - example

```
public class ArrayTest {
    public static void main(String[] args) {
      int a1[] = {10,34,56,23,67,87};
      int a2[];  // value is null
      int a3[] = new int[5];
      a2 = a1;  // a1 and a2 hold same array
/* use length to know the size of the array */
      for(int i=0;i<a1.length;i++)
            System.out.println(a1[i]);

}
```

Array of objects

- Array of objects is array of references to the objects as shown in this example

```
public class ArrayOfStringsDemo {
   public static void main(String[] args) {
     Test b[] = new Test[5];
     Test a[] = { new Test("Ramana"), new Test("Surender"),
                  new Test("Hiresh"), new Test("Haritha") };
       for (int i = 0; i < a.length; i++)  a[i].show();
   }
}
public class Test {
   private String name;
    public Test(String s) {
        name = s;
    }
    public void show() {
       System.out.println(name);
    }
}
```

Arrays and *for-each* Loop

```
1   public class ArrayOperations {
2       public static void main(String args[]){
3
4           String[] names = new String[3];
5
6           names[0] = "Blue Shirt";
7           names[1] = "Red Shirt";
8           names[2] = "Black Shirt";
9
10          int[] numbers = {100, 200, 300};
11
12          for (String name:names){
13              System.out.println("Name: " + name);
14          }
15
16          for (int number:numbers){
17              System.out.println("Number: " + number);
18          }
19      }
20  }
```

Arrays are objects.
Array objects have a final field length.

Inheritance

Inheritance

- Inheritance denotes Specialization.
- A *subclass* is a class that extends another class. A subclass inherits state and behavior from all of its ancestors (a.k.a super class(es)).
- A subclass inherits variables and methods from its superclass and all of its ancestors. The subclass can use these members as is, or it **can hide** the member variables or **override** the methods.

Constructors in inheritence

- While creating subclass objects super class default constructor is automatically invoked
- To invoke arguemented constructor of super class use super() with arguments
- super() should be first statement in subclass constructor

Overriding

- Overriding a method means replacing the superclass's implementation of a method with one of your own. The signature must be identical
- When a method is overridden it means both the signature and return type are SAME as in the superclass.
- If two methods differ only in return type it is an ERROR and the compiler will reject the class.

Overriding (cont…)

- Overriding methods can have their own access specifiers. A subclass can change the access of a superclass's methods, but ONLY to provide MORE access.

- For example

```
class Base {
    Protected void show() {
    }
}
Class Derived extends Base{
    Public void show() {  //this is valid
    }
}
```

Overriding (cont…)

- The Overriding method can be made final but not the method being overridden.
- Overriding method's throws clause CAN BE different from that of the superclass method's as long as every exception type listed in the overriding method is the same or a subtype of the exceptions listed in the superclass's method.
- An overriding method can have **NO** throws clause though the method in superclass has.
- Static method **CANNOT** be overridden.

Polymorphism

- Super class reference variable can hold sub class object
- When a overriden method is invoked on super class reference variable, the method of the object held by it is invoked

Polymorphism

Example :

```
class  Person  {
        public void display(){   system.out.println("Person");    }
}

class  Employee extends Person  {
        public void display(){   system.out.println("Employee Data");    }
}

class  Student extends Person  {
        public void display(){   system.out.println("Student Data");    }
}
```

Polymorphism

Example :

```
Person p ;

p  = new Person();
p.display();                    //  output :  Person

p = new employee();
p.display();                    // output  :  Employee Data

p = new Student();
p.display();                    // output : Student Data
```

Abstract methods & classes

- Methods whose design is not complete are called abstract methods
- Ex: public abstract void printIt(int x, String y);

- Class having abstract methods should be declared as abstract class
- Abstract class cannot be instantiated (cannot create objects)
- Abstract classes are for subclassing
- A class declared as abstract need not have abstract methods

final class

- Class declared as final cannot be extended
- final class cannot have abstract methods
- Ex:

```
public final class LastOne {
    --------
    --------

}
```

Interfaces

- The fundamental unit of OO design is the *type.*
- *Interfaces* define types in an abstract form as a collection of methods.
- *Interfaces* contain no implementation and you cannot create instances of an interface.
- Classes can expand their own types by implementing interfaces.

Interfaces (contd…)

- Classes can implement more than one interface.

- In a given class, the classes that are extended and the interfaces that are implemented are collectively called the *supertypes*, the new class is a *subtype*.

Interface example

- An example of a simple interface :

```
public interface Comparable {
 int compareTo(Object o);
}
```

Declarations

- An *interface* is declared using the keyword interface, giving the interface a name and listing the interface members between braces.

- An interface can have
  1. Constants
  2. Methods (only signature)
  3. Default methods
  4. Static methods

- Interface members are implicitly *public*

Interface constants

- An interface can declare named constants.
- These constants are *implicitly public, static, and final*.
- ```
  interface Verbose   {
     int SILENT = 0;
     int NORMAL = 1;
     int VERBOSE = 3;
  }
  ```
  ```

Interface methods

- The methods declared in an interface are implicitly abstract and public, no other modifiers are permitted
- Interfaces can have static methods which can be called using interface name
- Interfaces can also have fully implemented methods identified as default methods(java 8 onwards)

Implementing interfaces

- A class can implement one or more interfaces using *implements* keyword

```
public class Employee implements Comparable {
        public int compareTo(Object o ){
                // Implementation details

        }
}
```

- Default methods are fully implemented methods

Object class

- Every java class extends Object class either directly or indirectly
- Object class provides useful methods required in every class
- Some of the methods need to be overwritten
- If a class does not extend any class, it automatically extends Object
- Every java objet is instanceof Object

Object class

Some Object class methods

boolean equals (Object obj)

Decides whether two objects are meaningfully equivalent.

void finalize( )

Called by garbage collector when the garbage collector sees that the object cannot be referenced

int hashCode( )

Returns a hashcode int value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet

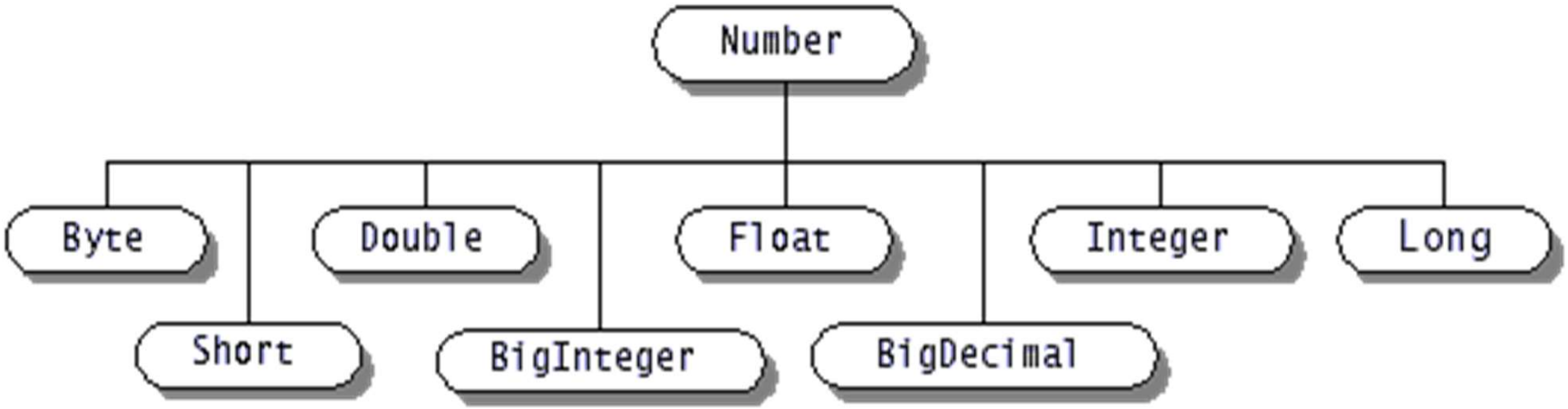
String toString( )

Returns a “text representation” of the object

Used by System.out.println(object)

Wrapper classes

- Wrapper classes available to create objects for all primitive types
- These classes provide methods to convert wrap primitive types and also convert back to primitives



- Also available, Boolean and Character classes

Wrapper classes

- Designed to convert primitives into objects
- Wrapper objects are immutable
- provide methods for conversion of primitives to/from String objects to different bases
- All wrapper class names map to primitives they represent except Integer and Character
- Byte, Short, Integer, Long, Float, Double are sub classes of Number
- constructors overloaded to take primitives as well as their String representation

Wrapper classes

- Common methods of wrapper classes

Methods of Number

```
byte    byteValue( )
short   shortValue( )
int     intValue( )
long    longValue( )
float   floatValue( )
double  doubleValue( )
```

Character

```
char charValue( )
```

Boolean

```
boolean booleanValue( )
```

Auto Boxing Unboxing

- Before Java 5, wrapping and unwrapping was done explicitly
- For example to perform arithmatic on a wrapped value involved unwrapping and re-wrapping after operation

```
Integer  x = new Integer(45);
int y = x.intValue( );
y = y +10 ;
x = new Integer(y) ;
```

- java 5 provides auto boxing / unboxing
- In that wrapping and unwrapping done automatically based on the operation

```
Integer  x = 45;
x  = x +10 ;
x++;   // and so on
```

Auto Boxing Unboxing

All these are possible now !!!

```
Integer  x = 36;    // wrap it
x++;                // unwrap and re-wrap

List l = new ArrayList();
 l.add(0,36);       //wrap and add

Integer a = 30;    //wrap
Integer b = 20;    //wrap
Integer c = a + b;     //unwrap, add, wrap the sum

if (a.equals(30) )    //unwrap and compare
```

Math class

- Math class provides many arithmatic, trignometric and logarithmic methods
- All these methods are static
- It is not possible to create Math class object
- Example methods:

  static double sqrt(double)

  static double sin(double)

  static double random( )

Working with Local Date and Time

- The *java.time* API defines two classes for working with local dates and times (without a time zone):
  - *LocalDate*:
    - Does not include time
    - A year-month-day representation
    - *toString* – ISO 8601 format (YYYY-MM-DD)
  - *LocalTime*:
    - Does not include date
    - Stores hours:minutes:seconds.nanoseconds
    - *toString* – (HH:mm:ss.SSSS)

`LocalDate`: Example

```
import java.time.LocalDate;
import static java.time.temporal.TemporalAdjusters.*;
import static java.time.DayOfWeek.*;
import static java.lang.System.out;

public class LocalDateExample {

  public static void main(String[] args) {
    LocalDate now, bDate, nowPlusMonth, nextTues;
    now = LocalDate.now();
    out.println("Now: " + now);
    bDate = LocalDate.of(1995, 5, 23); // Java's Birthday
    out.println("Java's Bday: " + bDate);
    out.println("Is Java's Bday in the past? " + bDate.isBefore(now));
    out.println("Is Java's Bday in a leap year? " + bDate.isLeapYear());
    out.println("Java's Bday day of the week: " + bDate.getDayOfWeek());
    nowPlusMonth = now.plusMonths(1);
    out.println("The date a month from now: " + nowPlusMonth);
    nextTues = now.with(next(TUESDAY));
    out.println("Next Tuesday's date: " + nextTues);
  }
}
```

`next` method

`TUESDAY`

`LocalDate` objects are immutable – methods return a new instance.

Packages

- Packages are convenient ways of grouping related classes according to their functionality, usability as well as category they should belong to.
- Classes under different packages **CAN HAVE** same names.
- Packaging help us to avoid class name collision when we use the same class name as that of others

How to create Packages?

1. Suppose we have a file called **HelloWorld.java**, and we want to put this file in a package **world** then add the package definition in the top of the file as shown below…

```
// only comment are allowed before this definition
package world;
public class HelloWorld {
//…
}
```

2. Create subdirectories to represent package hierachy of the class. In our case, we have the **world** package, which requires only one directory. So, we create a directory **world** and put our HelloWorld.java into it.

How to use Packages?

- There are 2 ways in order to use the public classes stored in package.

1. Declare the fully-qualified class name. For example,

```
world.HelloWorld hw = new world.HelloWorld();
world.moon.HelloMoon hm = new world.moon.HelloMoon();
String holeName = helloMoon.getHoleName();
```

2. Use an "import" keyword:

```
import world.*;
import world.moon.*;
HelloWorld helloWorld = new HelloWorld();
HelloMoon helloMoon = new HelloMoon();
```

Using Access Control

- There are four access levels that can be applied to data fields and methods. The following table illustrates access to a field or a method marked with the access modifier in the left column.

Modifier (keyword)	Same Class	Same Package	Subclass in Another Package	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes *	
public	Yes	Yes	Yes	Yes

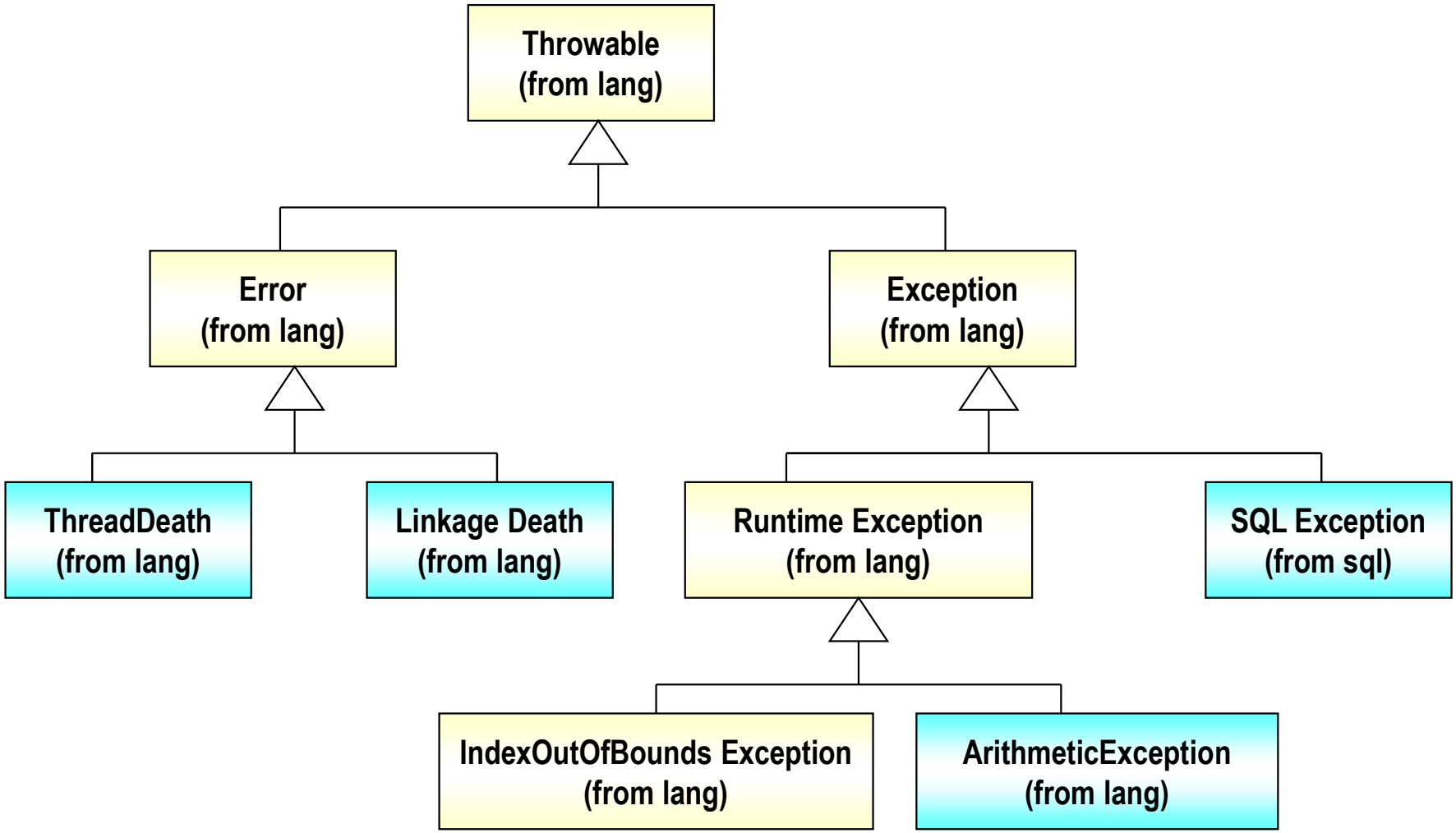
Exceptions

Exception Handling

- Exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions

- Exceptions can occur when
  - The file you try to open does not exist
  - The network connection is disrupted
  - Operands being manipulated are out of prescribed ranges
  - The class files you are interested in loading are missing

Exception class hirarchy



Exception Handling

- Checked Exceptions
  - Extends the java.lang.Exception class.
  - Needs to be caught or specified.
- Unchecked Exceptions
  - Extends the java.lang.RuntimeException class
  - Need not be caught or specified.

Exception Handling

- Methods should either catch or specify all checked exceptions that can be thrown within the scope of that method
- A method can catch an exception by providing an exception handler for that type of exception
- If a method chooses not to handle an exception, the method must specify that it can throw that exception
- Callers of a method must know about the exceptions that a method can throw

Dealing with Exceptions

- Three components of an exception handler

  try, catch, and finally blocks

try Block

- Enclose the statements that might throw an exception within a try block
- Defines the scope of any exception handlers

catch Block

- Associate exception handlers with a try block by providing one or more catch blocks directly after the try block

finally Block

- Allows the method to clean up after itself regardless of what happens within the try block

Collection classes

Collections

Basic operations of collections

- add objects to collection
- remove objects from collection
- search for an object in collection
- retrieve object from collection
- iterate through collection

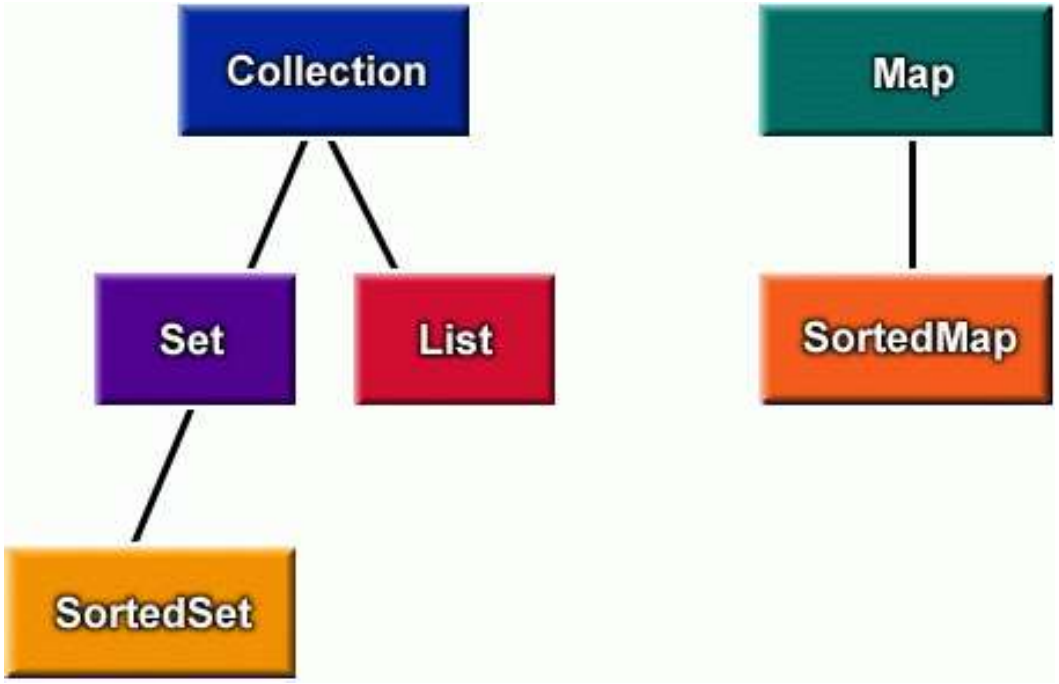
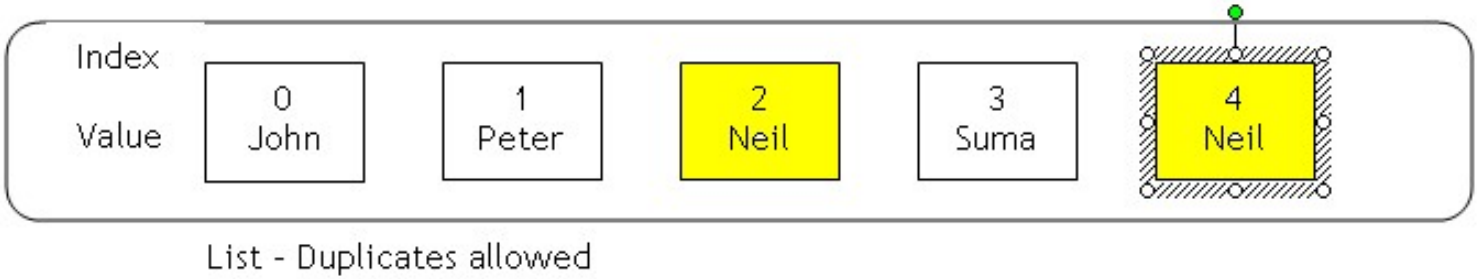
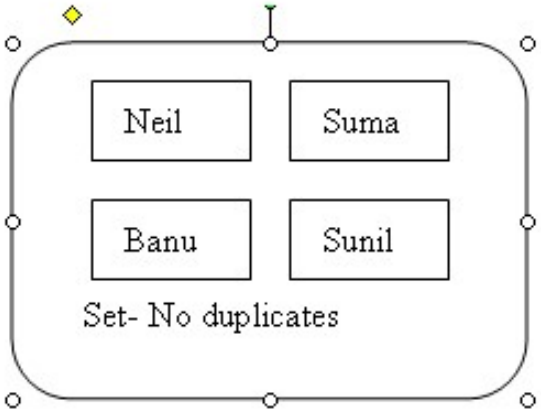
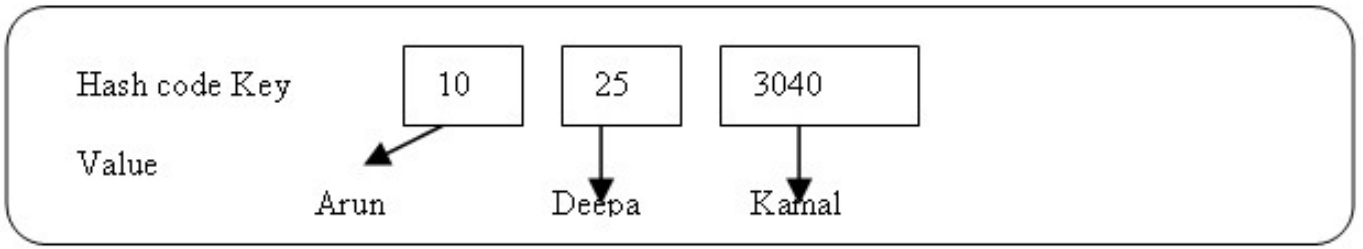


Illustration of List, Set and Map



List - Duplicates allowed





Hash Map - Key generated from RollNo

Collections

interface java.util.Collection

- root interface in the collections hierarchy
- extended by List, Set, Queue
- methods

```
boolean add(Object)
boolean remove(Object)
boolean addAll(Collection)
boolean removeAll(Collection)
Object[ ]  toArray( )
boolean contains(Object)
Iterator iterator( )
int size( )
```

Collections

interface java.util.List

- represents ordered collection
- allows duplicates
- implemented by ArrayList, Vector, LinkedList
- additional methods

```
void add(int index, Object)
 Object set(int index, Object)
boolean remove(int index)
boolean addAll(int index, Collection)
Object get(int index)
ListIterator listIterator( )
```

Collections

interface java.util.Set

- represents unordered collection
- duplicates not allowed
- extended by SortedSet

Collections

interface java.util.Map

- represents key-value pairs
- Not part of Collection hierarchy
- extended by SortedMap
- implemented by HashTable, HashMap
- methods

    boolean containsKey(Object key)

    boolean containsValue(Object value)

    Object get(Object key)

    Set keySet( )

    Collection values( )

    Object put(Object key, Object value)

    Object remove(Object key)

Collections

Classes

- ArrayList : Resizable array implementation of List
  Implements RandomAccess interface
- Vector : Same as ArrayList but threadsafe (legacy class)
- LinkedList : Linked list implementation of List
  provides add, remove at beginning or end
- HashSet : Unsorted, unordered implementation of Set
  uses hashCode( )
- LinkedHashSet : ordered version of HashSet
- TreeSet : implementation of SortedSet (elements sorted)
- HashMap : unsorted Map implementation
- Hashtable : same as HashMap but threadsafe (legacy class)
- TreeMap : implementation of SortedMap

ArrayList

- Is an implementation of the *List* interface
  - The list automatically grows if elements exceed initial size.
- Has a numeric index
  - Elements are accessed by index.
  - Elements can be inserted based on index.
  - Elements can be overwritten.
- Allows duplicate items

```
List partList = new ArrayList();
    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add(new Integer(4444)); // ArrayList auto grows
    System.out.println("First Part: " +
                       partList.get(0)); // First item
    partList.add(0, new Integer(5555)); // Insert at position 1
```

TreeSet: Implementation of *Set*

```
public class SetExample {
   public static void main(String[] args){
        Set set = new TreeSet();

        set.add("one");
        set.add("two");
        set.add("three");
        set.add("three"); // not added, only unique

        Iterator itr= set.iterator();
         while(itr.hasnext()){
             System.out.println("Item: " + itr.next());
        }
   }
}
```

Collections

java.util.Iterator interface

Used to iterate through all the elements of the collection

Methods

boolean hasNext( )

Object next( )

void remove( )

Enhanced for loop

- Iterating over collections looks cluttered

```
ArrayList lst = …….;
Iterator i = lst.iterator();
While( i.hasNext() )
  System.out.println(i.next());
```

- Using enhanced for loop we can do the same thing as

```
ArrayList lst = …….;
for (Object t: lst) )
    System.out.println(t);
```

TreeMap: Implementation of *Map*

```
public class MapExample {
   public static void main(String[] args){
       Map partList = new TreeMap();
       partList.put("S001", "Blue Polo Shirt");
       partList.put("S002", "Black Polo Shirt");
       partList.put("H001", "Duke Hat");

       partList.put("S002", "Black T-Shirt"); // Overwrite value
       Set keys = partList.keySet();

       System.out.println("=== Part List ===");
       for (Object key:keys){
           System.out.println("Part#: " + key + " " +
                              partList.get(key));
       }
   }
}
```

Generic Collections

- Generic collections used to hold homogeneous data
- They are type safe
- No typecasting required while extracting elements

```
List<Emp> list = new ArrayList<Emp>();
 list.add(new Emp());
 Emp e = list.get(0);

 HashMap<String, Mammal> map =
  new HashMap<String, Mammal>();
  map.put("wombat", new Mammal("wombat"));
  Mammal w = map.get("wombat");
```

Ordering Collections

- The *Comparable* and *Comparator* interfaces are used to sort collections.
  - Both are implemented by using generics.
- Using the *Comparable* interface:
  - Overrides the *compareTo* method
  - Provides only one sort option
- The *Comparator* interface:
  - Is implemented by using the *compare* method
  - Enables you to create multiple *Comparator* classes
  - Enables you to create and use numerous sorting options

Comparable: Example

```
public class Student implements Comparable<Student>{
   private String name;
   private long id = 0;
   private double gpa = 0.0;

   public Student(String name, long id, double gpa){
      // Additional code here
   }
      // getters and setters

   public int compareTo(Student s){
       if(this.id < s.id)
                 return -1;
       if (this.id > s.id)
                  return 1;
       return 0;
   }
}
```

Comparable : Example

```
public class TestComparable {
  public static void main(String[] args){
    Set<Student> studentList = new TreeSet<Student>();

    studentList.add(new Student("Thomas Jefferson", 1111, 3.8));
    studentList.add(new Student("John Adams", 2222, 3.9));
    studentList.add(new Student("George Washington", 3333, 3.4));

    for(Student student:studentList){
      System.out.println(student);
    }
  }
}
```

Comparator - Example

```
class IdComp implements Comaparator<Student> {
    public int compare(Student a, Student b){
        return a.getId () – b.getId( );
    }
}

class NameComp implements Comaparator<Student> {
    public int compare(Student a,  Student b){
        return a.getName().compareTo( b.getName( ) );
    }
}
…….
TreeSet <Student>  ts1 =  new TreeSet<Student> ( new IdComp() );  //sorted on id
TreeSet <Student> ts2 =  new TreeSet<Student> ( new NameComp() );  // sorted on name
```

Collections class

- Collections class is used exclusively with static methods that operate on or return collections
- provides some convenience methods that are highly useful in working with Java collections
- Some of the methods of Collections:

  - **static <T> int binarySearch**(**List**,<T> T key) - Searches the list for the specified object using the binary search algorithm.
  - **static<T> void copy**(**List <T>** dest, **List <T>** src) - Copies all of the elements from one list into another
  - **static<T> void sort**(**List <T> list**) - Sorts the list into ascending order, according to the **natural ordering** of its elements
  - **static<T> void sort**(**List <T> list , Comparator<T> c**) - Sorts the list according to the order induced by the specified comparator
  - **static void swap**(**List <T> list , int I, int j**) - Swaps the elements at the specified positions in the list

SQL

Relating Multiple Tables

- **Each row of data in a table is uniquely identified by a primary key.**
- **You can logically relate data from multiple tables using foreign keys.**

Table name: DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting	(null)	1700

Table name: EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
100	Steven	King	90
101	Neena	Kochhar	90
102	Lex	De Haan	90
103	Alexander	Hunold	60
104	Bruce	Ernst	60
107	Diana	Lorentz	60
124	Kevin	Mourgos	50
141	Trenna	Rajs	50
142	Curtis	Davies	50

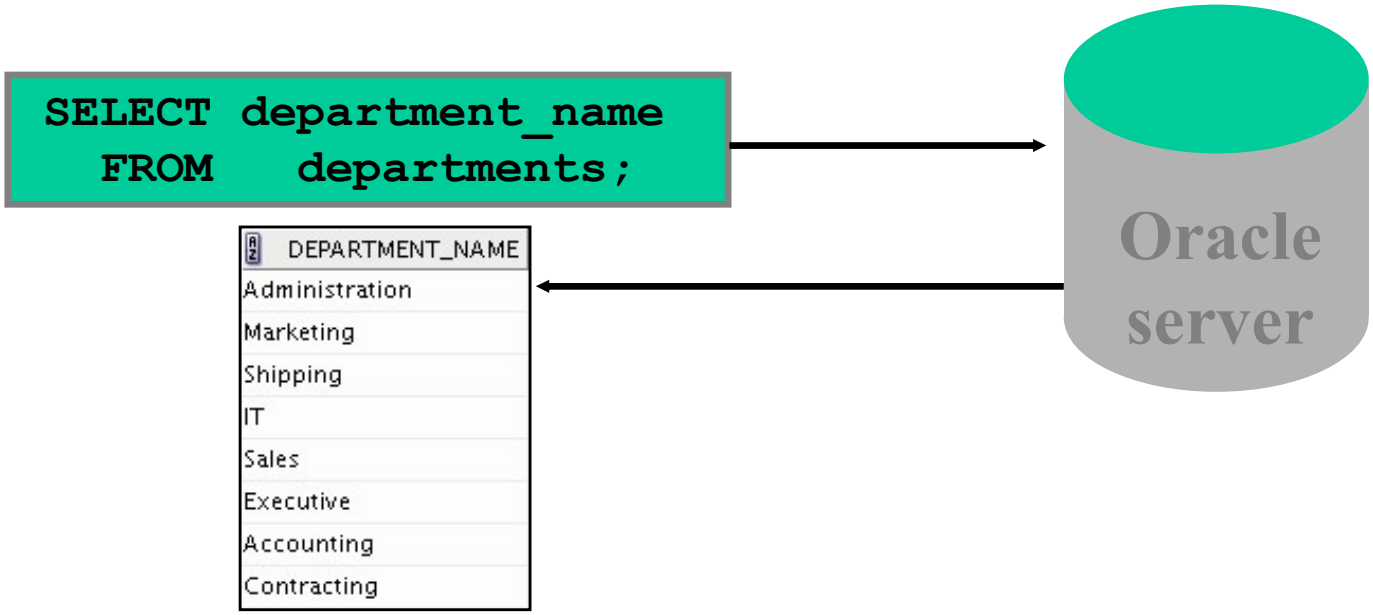
Primary key

Primary key

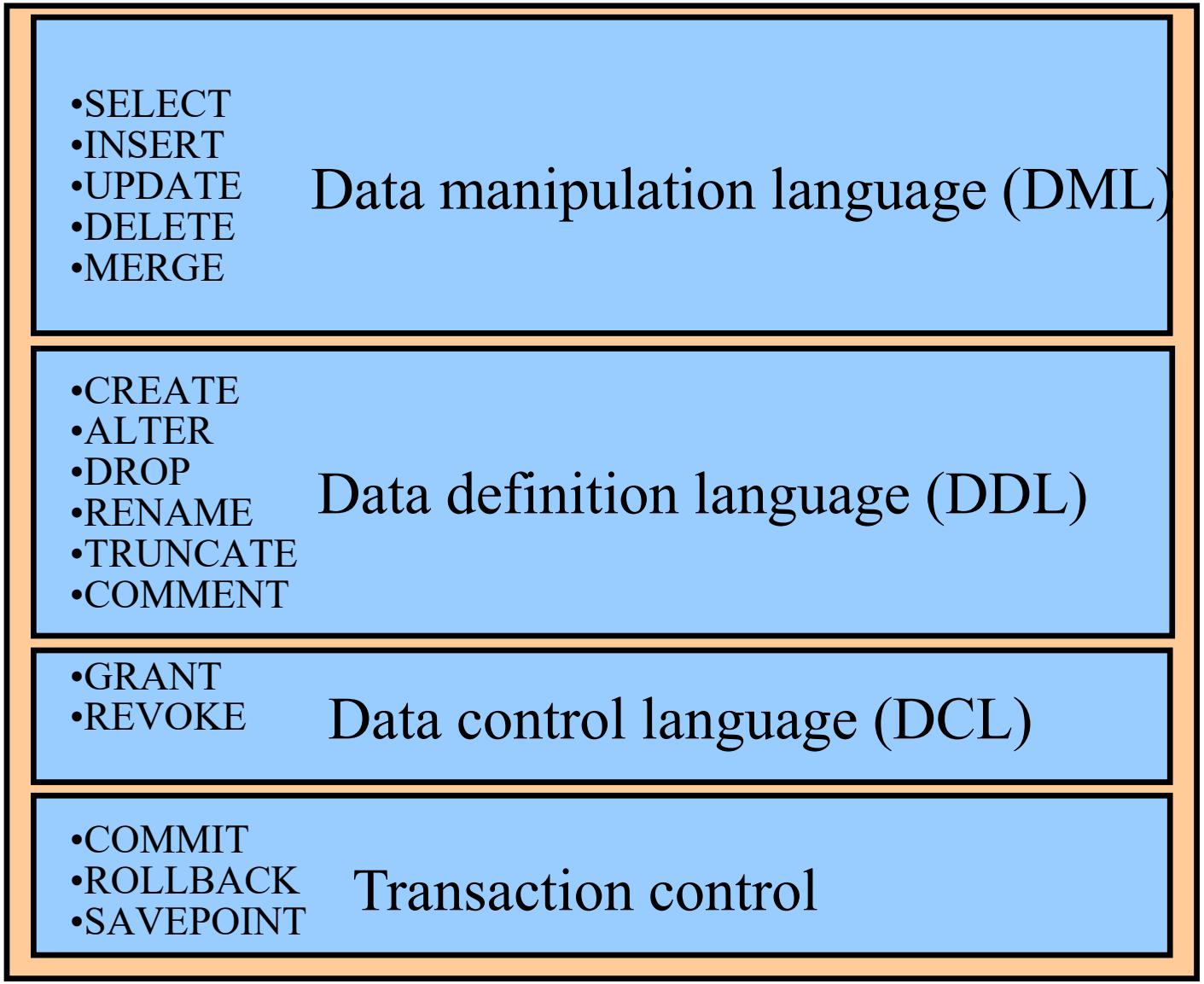
Foreign key

Using SQL to Query Your Database

- **Structured query language (SQL) is:**
  - **The ANSI standard language for operating relational databases**
  - **Efficient, easy to learn, and use**
  - **Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)**



SQL Statements



CREATE TABLE Statement

- You specify:
  - The table name
  - The column name, column data type, and column size

```
CREATE TABLE tableName   (
      column datatype [, ...]
      [Constraint specification
);
```

- To see the table structure
  - Describe tableName

Creating Tables

– Create the table:

```
CREATE   TABLE  ord
        ( ID  DECIMAL(3),
         quantity  DECIMAL(3)
         ) ;
```

Data Types

Java DB, Derby	Format
SMALLINT	
INTEGER	
DECIMAL(p,s) or NUMERIC(p,s)	
FLOAT	
FLOAT	
SMALLINT	
SMALLINT	
VARCHAR	Single qoutes
CHAR(1)	Single qoutes
DATE	yyyy-mm-dd, mm/dd/yyyy, dd.mm.yyyy
TIME	hh:mm[:ss] , hh.mm[.ss]
TIMESTAMP	yyyy-mm-dd hh:mm:ss[.nnnnnn]

Including Constraints

Constraints enforce rules at the table level.

Constraints prevent the deletion of a table if there are dependencies.

The following constraint types are valid:

- **NOT NULL**
- **UNIQUE**
- **PRIMARY KEY**
- **FOREIGN KEY**
- **CHECK**

CREATE TABLE: Example

```
CREATE   TABLE   customers

          ( customer_id          DECIMAL(6)  primary key
          , cust_first_name      VARCHAR(20)
               CONSTRAINT fname_nn NOT NULL
          , cust_last_name       VARCHAR(20)
               CONSTRAINT  lname_nn NOT NULL
          , cust_address         varchar(50)
          , city_code                  char(2)
          , language             VARCHAR(3)
          , territory            VARCHAR(30)
          , credit_limit         NUMERIC(9,2)
          , cust_email           VARCHAR(30)
          , account_mgr_id       NUMERIC(6)
          ,   CONSTRAINT  ck_credit_limit
                            CHECK (credit_limit <= 5000)
          ,   CONSTRAINT  city_fk
               foreign key(city_code)
                 references city(city_code)
    ) ;
```

Basic *SELECT* Statement

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM    table;
```

SELECT identifies the columns to be displayed.

FROM identifies the table containing those columns.

Selecting All Columns

```
SELECT  *
FROM   inventories ;
```

	PRODUCT_ID	WAREHOUSE_ID	QUANTITY_ON_HAND
1	3108	8	122
2	3110	8	123
3	3112	8	123
4	3117	8	124
5	3124	8	125
6	3127	8	125
7	3129	8	126
8	3134	8	149
9	3139	8	150
10	3140	8	150
11	3143	8	151

Selecting Specific Columns

```
SELECT  product_id, quantity_on_hand
FROM   inventories ;
```

	PRODUCT_ID	QUANTITY_ON_HAND
1	3108	122
2	3110	123
3	3112	123
4	3117	124
5	3124	125
6	3127	125
7	3129	126
8	3134	149
9	3139	150
10	3140	150
11	3143	151

Limiting the Rows That Are Selected

- **Restrict the rows that are returned by using the :**
- **WHERE clause**

```
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM   table
[WHERE condition(s)];
```

- **The WHERE clause follows the FROM clause.**

Using the *WHERE* Clause

```
SELECT   order_id,  order_date,  order_status
   FROM   orders
WHERE   order_status = 1 ;
```

	ORDER_ID	ORDER_DATE	ORDER_STATUS
1	2397	20-NOV-99 04.11.54.696211000 AM	1
2	2454	03-OCT-99 05.19.34.678340000 AM	1
3	2421	13-MAR-99 09.23.54.562432000 AM	1
4	2431	14-SEP-98 06.33.04.763452000 PM	1
5	2439	31-AUG-99 09.49.37.811132000 PM	1
6	2444	28-JUL-99 01.52.27.462632000 AM	1

Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN ...AND...	Between two values (inclusive)
IN(set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Using Comparison Operators

```
SELECT   order_id,  order_date
FROM   orders
WHERE   order_id <= 2400 ;
```

	ORDER_ID	ORDER_DATE
1	2354	15-JUL-00 05.48.23.234567000 AM
2	2355	26-JAN-98 10.52.51.962632000 PM
3	2356	26-JAN-00 10.52.41.934562000 PM
4	2357	09-JAN-98 09.49.44.123456000 AM
5	2358	09-JAN-00 06.33.12.654278000 AM
6	2359	09-JAN-98 11.04.13.112233000 AM

• • •

Range Conditions Using the *BETWEEN* Operator

- **Use the BETWEEN operator to display rows based on a range of values:**

```
SELECT   product_id,  quantity_on_hand
FROM   inventories
WHERE   product_id  BETWEEN  3100  AND  3108;
```

Lower limit **Upper limit**

	PRODUCT_ID	QUANTITY_ON_HAND
1	3108	122
2	3108	110
3	3108	194
4	3108	170
5	3108	146

Membership Condition Using the *IN* Operator

- **Use the IN operator to test for values in a list:**

```
SELECT   order_id,  order_mode,  order_status
FROM   orders
WHERE   order_id  IN  (2458, 2397, 2454) ;
```

	ORDER_ID	ORDER_MODE	ORDER_STATUS
1	2397	direct	1
2	2454	direct	1
3	2458	direct	0

Pattern Matching Using the *LIKE* Operator

Use the LIKE operator to perform wildcard searches of valid search string values.

Search conditions can contain either literal characters or numbers:

- **% denotes zero or many characters.**
- **_ denotes one character.**

```
SELECT   first_name
FROM     employees
WHERE    first_name LIKE 'S%' ;
```

Combining Wildcard Characters

- **You can combine the two wildcard characters (%, _) with literal characters for pattern matching:**

```
SELECT   last_name
FROM   employees
WHERE   last_name LIKE '_o%' ;
```

	LAST_NAME
1	Kochhar
2	Lorentz
3	Mourgos

- **You can use the ESCAPE identifier to search for the actual % and _ symbols.**

Using the *NULL* Conditions

Test for nulls with the IS NULL operator.

```
SELECT   order_ID,  order_status,  sales_rep_id
   FROM   orders
WHERE  sales_rep_id  IS  NULL;
```

	ORDER_ID	ORDER_STATUS	SALES_REP_ID
1	2355	8	(null)
2	2356	5	(null)
3	2359	9	(null)
4	2361	8	(null)
5	2362	4	(null)
6	2363	0	(null)

Defining Conditions Using the Logical Operators

Operator	Meaning
`AND`	Returns `TRUE` if *both* component conditions are true
`OR`	Returns `TRUE` if *either* component condition is true
`NOT`	Returns `TRUE` if the condition is false

Using the *AND* Operator

AND requires both the component conditions to be true:

```
SELECT   order_mode,  order_status,  customer_id
FROM   orders
WHERE   order_mode  = ' direct '
AND        customer_id = 103;
```

	ORDER_MODE	ORDER_STATUS	CUSTOMER_ID
1	direct	1	103
2	direct	4	103

Using the *OR* Operator

OR requires either component condition to be true:

```
SELECT   order_id,  order_status,  order_total
FROM   orders
WHERE   order_status = 0
            OR  order_total  >=  100000 ;
```

	ORDER_ID	ORDER_STATUS	ORDER_TOTAL
1	2458	0	70647.34
2	2354	0	46257
3	2434	8	242458.25
4	2361	8	120131.3
5	2363	0	10082.3
6	2367	10	144054.8
7	2369	0	11097.4
8	2375	2	103834.4
9	2385	4	295892
10	2388	4	282694.3
11	2399	0	25270.3

Using the *NOT* Operator

```
SELECT  order_id, order_status, order_total
FROM  orders
WHERE  order_status
          NOT  IN  (0,1,2,3) ;
```

	ORDER_ID	ORDER_STATUS	ORDER_TOTAL
1	2357	5	59872.4
2	2394	5	21863
3	2435	6	62303
4	2455	7	14087.5
5	2379	8	17848.2
6	2396	8	34930
7	2434	8	242458.25
8	2436	8	6394.8
9	2446	8	93570.57
10	2447	8	33893.6
11	2432	10	10523

Using the *ORDER BY* Clause

Sort the retrieved rows with the ORDER BY clause:

- **ASC: Ascending order, default**
- **DESC: Descending order**

The ORDER BY clause comes last in the SELECT statement:

```
SELECT   order_id,  order_date,  order_status
FROM   orders
ORDER BY   order_date ;
```

	ORDER_ID	ORDER_DATE	ORDER_STATUS
1	2442	27-JUL-90 11.52.59.662632000 PM	9
2	2445	28-JUL-90 03.04.38.362632000 AM	8
3	2418	21-MAR-96 05.48.21.862632000 AM	4
4	2357	09-JAN-98 09.49.44.123456000 AM	5

Group Functions

Group functions operate on sets of rows to give one result per group.

EMPLOYEES

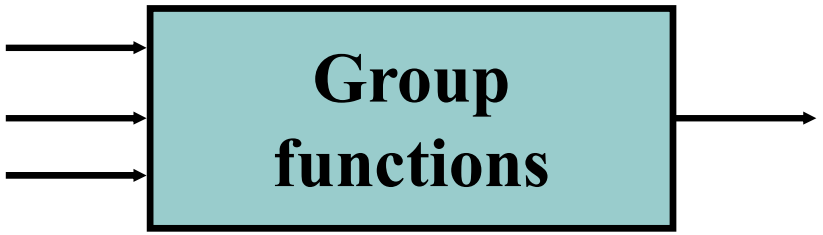
	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	110	12000
5	110	8300
6	90	24000
7	90	17000
8	90	17000
9	60	9000
10	60	6000
...		
18	80	11000
19	80	8600
20	(null)	7000

Maximum salary in EMPLOYEES table

MAX(SALARY)
24000

Types of Group Functions

- *AVG*
- *COUNT*
- *MAX*
- *MIN*
- *SUM*



Group Functions: Syntax

```
SELECT group_function(column), ...
FROM     table
[WHERE   condition]
[ORDER BY  column];
```

Using the *AVG* and *SUM* Functions

You can use AVG and SUM for numeric data.

You can use MIN and MAX for numeric, character, and date data types.

```
SELECT   AVG(order_total),  MAX(order_total),
         MIN (order_total), SUM( order_total)
FROM   orders;
```

	AVG(ORDER_TOTAL)	MAX(ORDER_TOTAL)	MIN(ORDER_TOTAL)	SUM(ORDER_TOTAL)
1	44628.44125	295892	5451	3570275.3

Using the *COUNT* Function

- **COUNT(*) returns the number of rows in a table:**

```
SELECT   count(*)
FROM   inventories
WHERE   warehouse_id = 8;
```

	COUNT(*)
1	186

- **COUNT(*expr*) returns the number of rows with non-null values for *expr*:**

```
SELECT   COUNT(sales_rep_id)
FROM   orders
WHERE   order_status <=3;
```

	COUNT(SALES_REP_ID)
1	19

Creating Groups of Data

EMPLOYEES

	DEPARTMENT_ID	SALARY	
1	10	4400	4400
2	20	13000	9500
3	20	6000	
4	50	2500	
5	50	2600	
6	50	3100	3500
7	50	3500	
8	50	5800	
9	60	9000	
10	60	6000	6400
11	60	4200	
12	80	11000	10033
13	80	8600	
...			
18	110	8300	
19	110	12000	
20	(null)	7000	

Average salary in the EMPLOYEES table

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	20	9500
3	90	19333.333333333333...
4	110	10150
5	50	3500
6	80	10033.333333333333...
7	10	4400
8	60	6400

Creating Groups of Data: *GROUP BY* Clause Syntax

•You can divide rows in a table into smaller groups by using the GROUP BY clause.

```
SELECT     column, group_function(column)
FROM       table
[WHERE     condition]
[GROUP BY group_by_expression]
[ORDER BY column];
```

Using the *GROUP BY* Clause

- **All the columns in the SELECT list that are not in group functions must be in the GROUP BY clause.**

```
SELECT   warehouse_id,  AVG(quantity_on_hand)
FROM   inventories
GROUP BY   warehouse_id ;
```

	WAREHOUSE_ID	AVG(QUANTITY_ON_HAND)
1	1	152.30555555555555555555555555555555555556
2	2	161.65536723163841807909604519774011299 4
3	3	151.08333333333333333333333333333333333333
4	4	136.33027522935779816513761467889908256 9
5	5	113.76315789473684210526315789473684210 5
6	6	98.35096153846153846153846153846153846154
7	7	85.27358490566037735849056603773584905 66
8	8	72.48387096774193548387096774193548387097
9	9	57.4765625

Using the *GROUP BY* Clause

- The *GROUP BY* column does not have to be in the *SELECT* list.

```
SELECT   AVG(order_total)
FROM   orders
GROUP BY   order_status ;
```

	AVG(ORDER_TOTAL)
1	36613.683333333333333333333333333333333
2	41017.96
3	43866.746666666666666666666666666666667
4	34772.38
5	83876.466666666666666666666666666666667
6	55182.434666666666666666666666666666667
7	36222.094285714285714285714285714285714
8	11205.7
9	28134.223333333333333333333333333333333
10	58939.925
11	28913.066666666666666666666666666666667

Restricting Group Results

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	2500
5	50	2600
6	50	3100
7	50	3500
8	50	5800
9	60	9000
10	60	6000
11	60	4200
12	80	11000
13	80	8600
...		
18	110	8300
19	110	12000
20	(null)	7000

The maximum salary per department when it is

	DEPARTMENT_ID	MAX(SALARY)
1	20	13000
2	90	24000
3	110	12000
4	80	11000

Restricting Group Results with the *HAVING* Clause

When you use the HAVING clause, the server restricts groups as follows:

Rows are grouped.

The group function is applied.

Groups matching the HAVING clause are displayed.

```
SELECT     column, group_function
FROM       table
[WHERE     condition]
[GROUP BY group_by_expression]
[HAVING    group_condition]
[ORDER BY column];
```

Using the *HAVING* Clause

```
SELECT   warehouse_id,  AVG(quantity_on_hand)
FROM   inventories
GROUP BY   warehouse_id
HAVING  MAX  (quantity_on_hand) > 130 ;
```

	WAREHOUSE_ID	AVG(QUANTITY_ON_HAND)
1	1	152.305555555555555555555555555555555556
2	6	98.35096153846153846153846153846153846154
3	2	161.655367231638418079096045197740112994
4	4	136.330275229357798165137614678899082569
5	5	113.763157894736842105263157894736842105
6	8	72.48387096774193548387096774193548387097
7	3	151.083333333333333333333333333333333333
8	7	85.2735849056603773584905660377358490566
9	9	57.4765625

Using the *HAVING* Clause

```
SELECT    job_id, SUM(salary) PAYROLL
FROM      employees
WHERE     job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING    SUM(salary) > 13000
ORDER BY SUM(salary);
```

	JOB_ID	PAYROLL
1	IT_PROG	19200
2	AD_PRES	24000
3	AD_VP	34000

Data Manipulation Language

A DML statement is executed when you:

- **Add new rows to a table**
- **Modify existing rows in a table**
- **Remove existing rows from a table**

A *transaction* consists of a collection of DML statements that form a logical unit of work.

Adding a New Row to a Table

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

70	Public Relations	100	1700

New row

Insert new row into the DEPARTMENTS table.

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	70	Public Relations	100	1700
2	10	Administration	200	1700
3	20	Marketing	201	1800
4	50	Shipping	124	1500
5	60	IT	103	1400
6	80	Sales	149	2500
7	90	Executive	100	1700
8	110	Accounting	205	1700
9	190	Contracting	(null)	1700

INSERT Statement Syntax

- **Add new rows to a table by using the INSERT statement:**

```
INSERT INTO table [(column [, column...])]
VALUES      (value [, value...]);
```

- **With this syntax, only one row is inserted at a time.**

Inserting New Rows

Insert a new row containing values for each column.

List values in the default order of the columns in the table.

Optionally, list the columns in the INSERT clause.

```
INSERT   INTO   order_items (order_id,
line_item_id,  product_id,  unit_price,  quantity)
VALUES   (2355, 1, 3108, 46, 200) ;
```

Enclose character and date values within single quotation marks.

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	60
104	Bruce	Ernst	6000	103	(null)	60
107	Diana	Lorentz	4200	103	(null)	60
124	Kevin	Mourgos	5800	100	(null)	50

Update rows in the EMPLOYEES table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	80
104	Bruce	Ernst	6000	103	(null)	80
107	Diana	Lorentz	4200	103	(null)	80
124	Kevin	Mourgos	5800	100	(null)	50

UPDATE Statement Syntax

- **Modify existing values in a table with the UPDATE statement:**

```
UPDATE      table
SET         column = value [, column = value, ...]
[WHERE      condition];
```

- **Update more than one row at a time (if required).**

Updating Rows in a Table

- **Values for a specific row or rows are modified if you specify the WHERE clause:**

```
UPDATE   inventories
SET   warehouse_id  =  7
WHERE   product_id  =  3108 ;
```

```
1 rows updated
```

- **Values for all the rows in the table are modified if you omit the WHERE clause**:

```
UPDATE   inventories
SET   warehouse_id  =  7 ;
```

- **Specify SET *column_name*= NULL to update a column value to NULL.**

DELETE Statement

- **You can remove existing rows from a table by using the DELETE statement:**

```
DELETE [FROM]   table
[WHERE   condition];
```

Deleting Rows from a Table

- **Specific rows are deleted if you specify the WHERE clause**:

```
DELETE   FROM   runreport
WHERE   comments  =  ' Editing Report ' ;
```

- **All rows in the table are deleted if you omit the WHERE clause:**

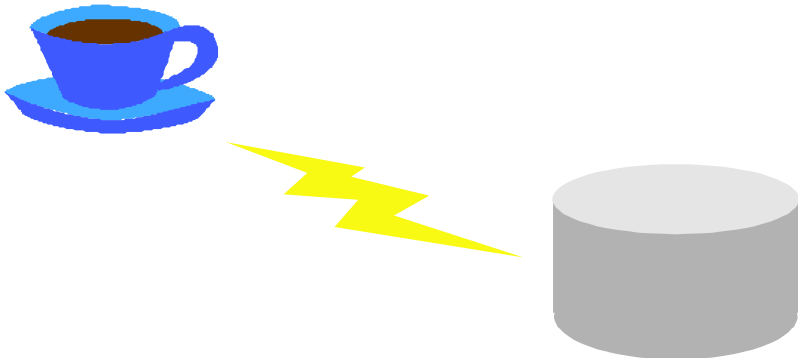
```
DELETE  FROM  copy_emp;
```

JDBC

Introduction to JDBC

- JDBC is a standard interface for connecting to relational databases from Java.

- The JDBC classes and interfaces are in the **java.sql** package.



java.sql package

Driver

DriverManager

DriverPropertyInfo

Connection

Statement

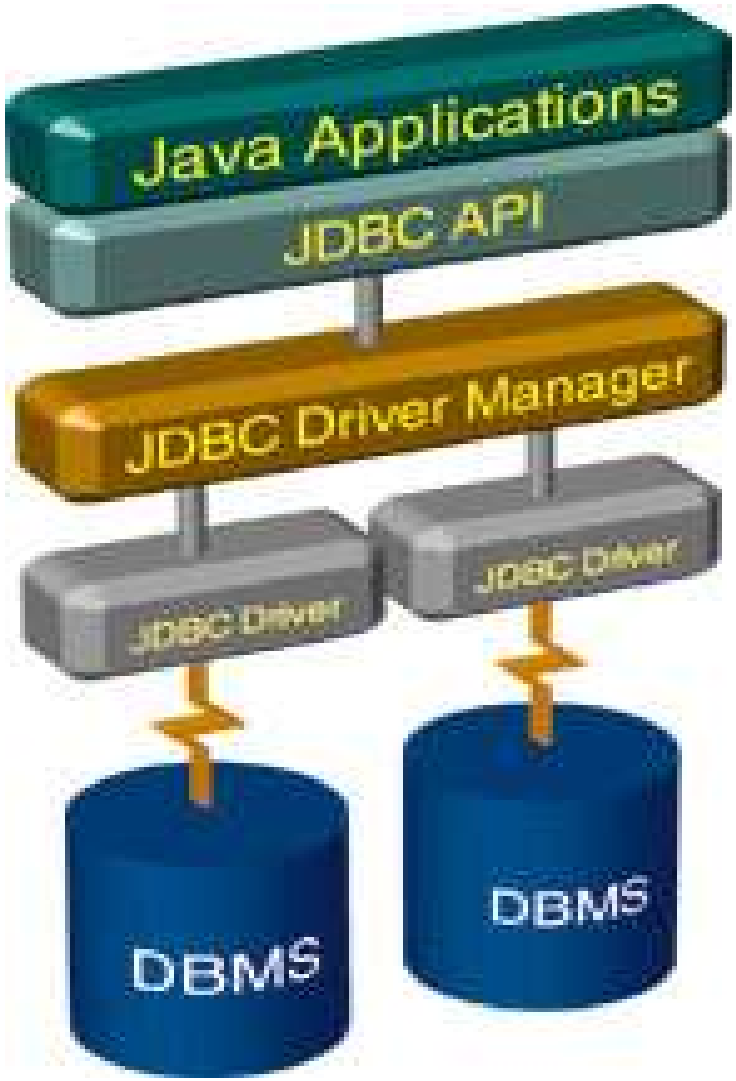
PreparedStatement

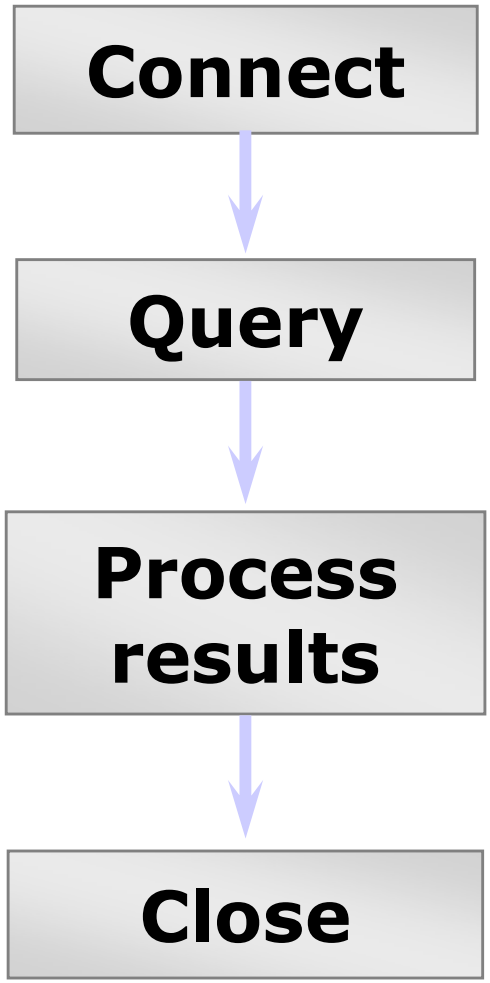
ResultSet
RowSet

SQLException
SQLWarning

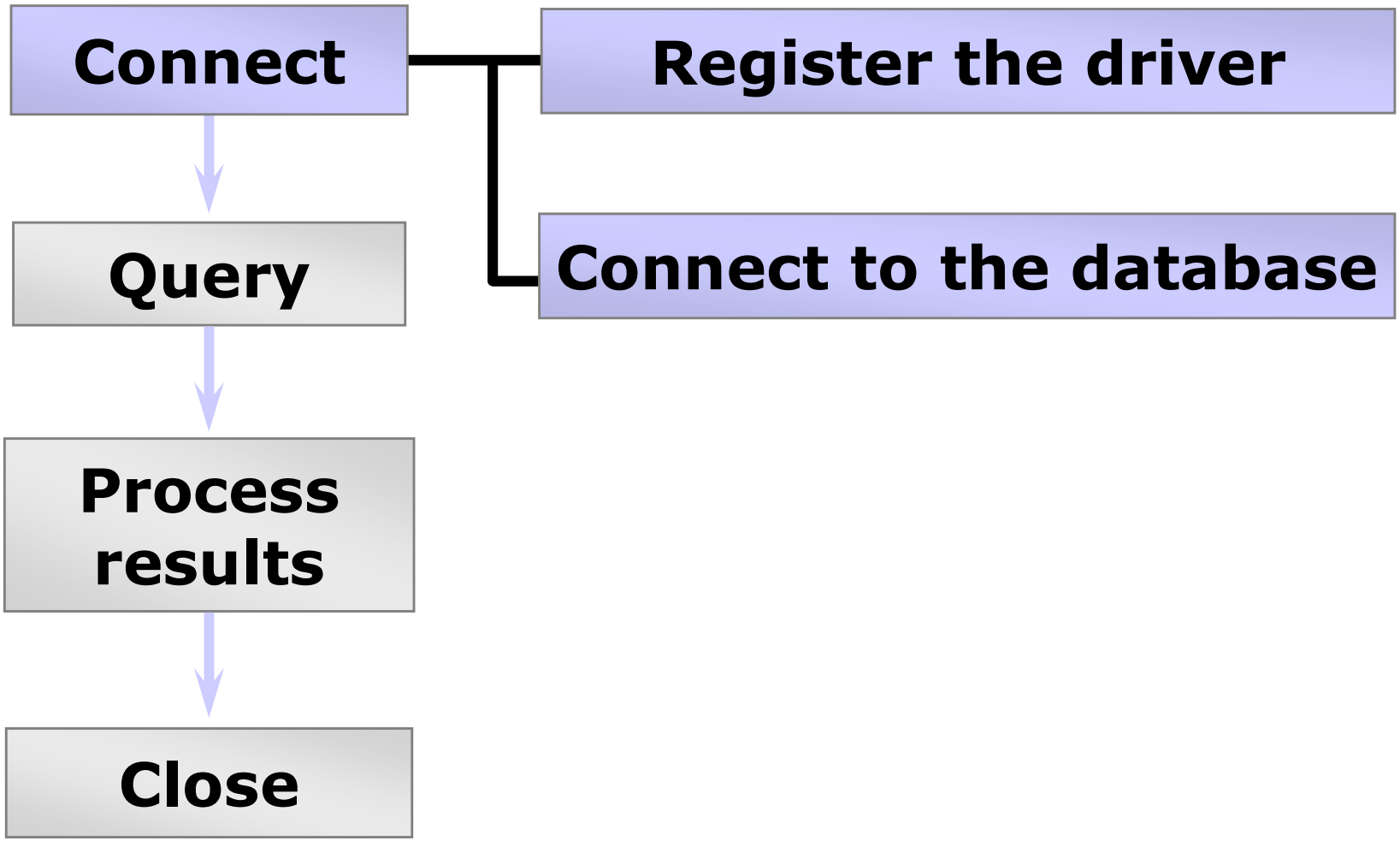
DatabaseMetaData
ResultSetMetaData

Architecture & Querying with JDBC



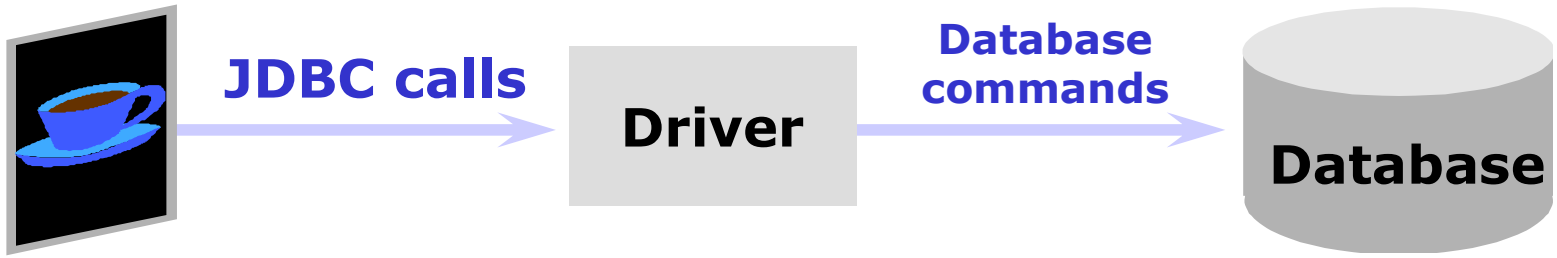


Step 1: Connect



Connect: **A JDBC Driver**

- A JDBC driver is an interpreter that translates JDBC method calls to vendor-specific database commands.
- JDBC driver Implements interfaces in `java.sql`



DriverManager class

All methods are static and the class does not have a constructor

Methods

Connection getConnection(String url)

Connection getConnection(String url , String user , String password)

Connection class

Methods

```
Statement createStatement( )
PreparedStatement prepareStatement( String sql )
void close ( )
DatabaseMetaData getMetaData( )
void setAutoCommit( boolean commit )     // default true
boolean getAutoCommit( )
void commit( )
void rollback( )
```

Setting up database connection

Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”)

Connection con = DriverManager.getConnection(url)

URL format

jdbc: <sub protocol > : <subname related to database>

example

jdbc : odbc : student

jdbc : ids : //www.test.com:90/conn?dbtype=odbc&dsn=student

jdbc:derby://localhost:1527/ramanadb

Creating Connection

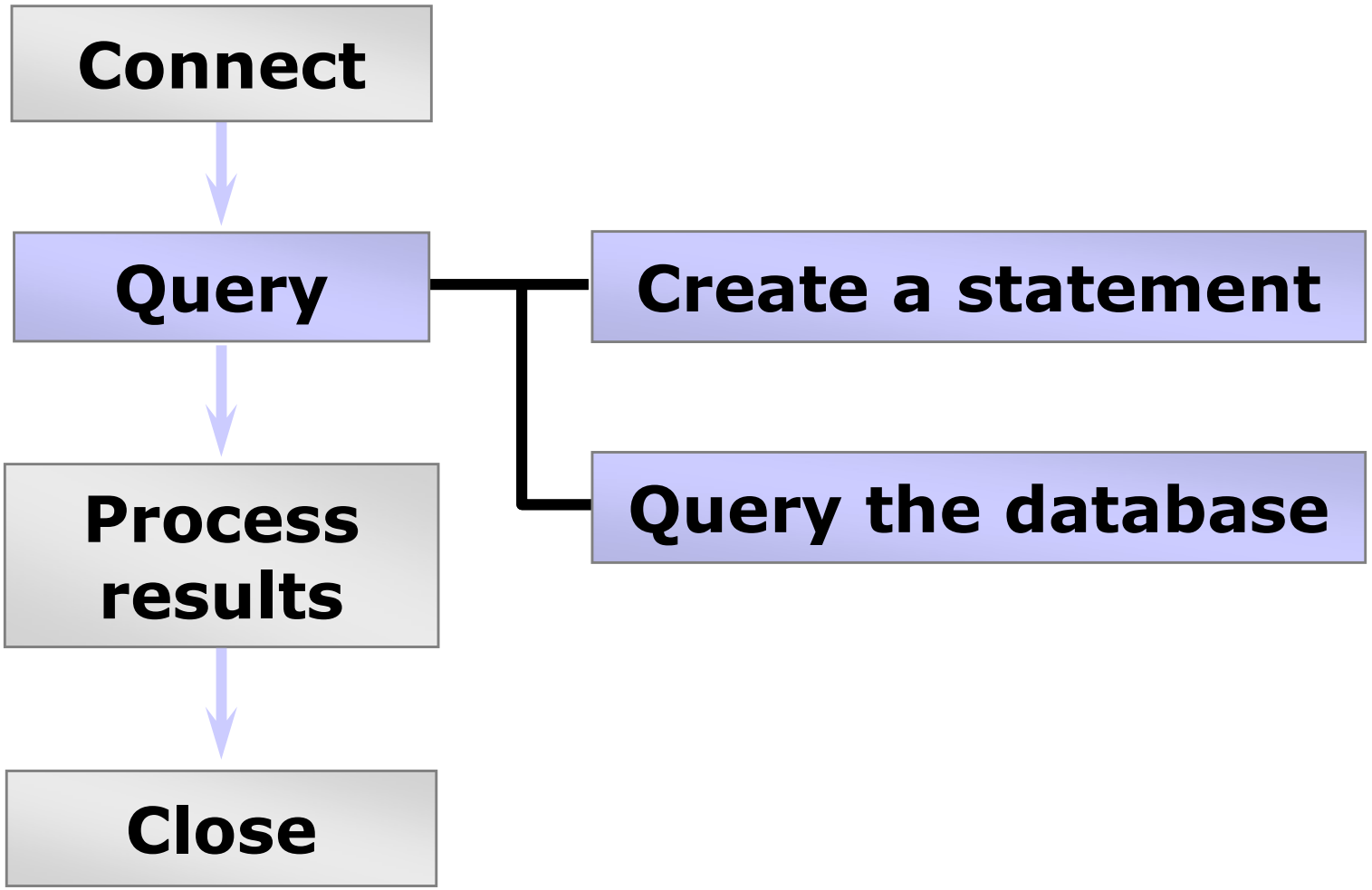
1. **Register the driver (prior to java 5. Not required now)**

```
Class c = Class.forName( " org.apache.derby.jdbc.ClientDriver ");
```

2. **Connect to the database**

```
Connection conn = DriverManager.getConnection(
        "jdbc:derby:codejava/webdb ",
        "user",
        "pwd");
```

Step 2: Query


Connect
Query
Create a statement
Query the database
Process results
Close

Statements

types of statements

- **Statement**
- **PreparedStatement**
- **CallableStatement**

All these are implemented as classes

Statement interface

Methods

ResultSet executeQuery(String query)

int executeUpdate(String sql)

boolean execute(String sql)

ResultSet getResultSet ( )

int getUpdateCount( )

Method	Returns	Used for
`executeQuery(sqlString)`	`ResultSet`	`SELECT` statement
`executeUpdate(sqlString)`	`int` (rows affected)	`INSERT`, `UPDATE`, `DELETE`, or a DDL
`execute(sqlString)`	`boolean` (true if there was a `ResultSet`)	Any SQL command or commands

Example code for Statement

```
Statement   stmt ;

private void runStatement()  throws SQLException {

   Class.forName ("jdbc.odbc.JdbcOdbcDriver");

   Connection con = DriverManager.getConnection(“jdbc:odbc:dsn” );

   String sql  = ”select name , salary from emp where  empno = 3010 ” ;

    stmt = con.createStatement(  ) ;

   ResultSet rs = stmt.executeQuery( sql ) ;

}
```

Statement methods

1. **Create an empty statement object**

```
Statement stmt = conn.createStatement();
```

2. **Execute the statement**

```
ResultSet rset = stmt.executeQuery(statement);
int count = stmt.executeUpdate(statement);
boolean isquery = stmt.execute(statement);
```

Statement methods: **Examples**

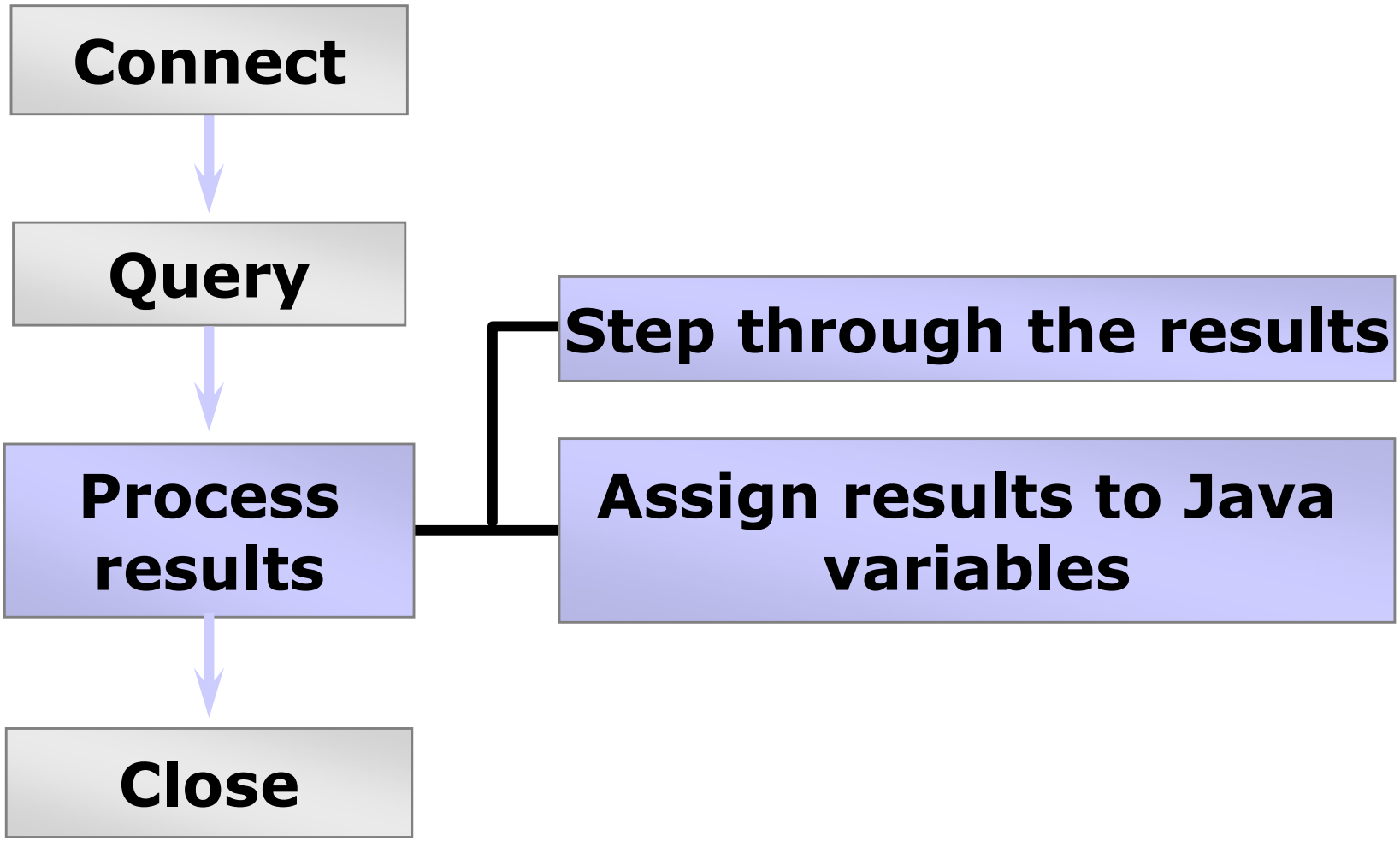
Execute a select statement

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery
  ("select NAME, VERTICAL from STUDENT");
```

Execute a delete statement

```
Statement stmt = conn.createStatement();
int rowcount = stmt.executeUpdate
  ("delete from STUDENT   where  ID = 1000");
```

Step 3: Process the Results



Using a `ResultSet` Object

```
String query = "SELECT * FROM Employee";
ResultSet rs = stmt.executeQuery(query);
```

ResultSet cursor →

The first `next()` method invocation returns `true`, and `rs` points to the first row of data.

rs.next() →	110	Troy	Hammer	1965-03-31	102109.15
rs.next() →	123	Michael	Walton	1986-08-25	93400.20
rs.next() →	201	Thomas	Fitzpatrick	1961-09-22	75123.45
rs.next() →	101	Abhijit	Gopali	1956-06-01	70000.00
rs.next() →	*No data*				

The last `next()` method invocation returns `false`, and the `rs` instance is now null.

ExecuteQuery() - example

```
1   package com.example.text;
2
3   import java.sql.DriverManager;
4   import java.sql.ResultSet;
5   import java.sql.SQLException;
6   import java.util.Date;
7
8   public class SimpleJDBCTest {
9
10     public static void main(String[] args) {
11         String url = "jdbc:oracle:thin@localhost:1521:xe";
12         String username = "hr";
13         String password = "hr";
14         String query = "SELECT * FROM Employee";
15         try {
             1 Connection con =
16               DriverManager.getConnection (url, username, password);
17               Statement stmt = con.createStatement ();
18               ResultSet rs = stmt.executeQuery (query) ;
```

ExecuteQuery() - example

```
19              while (rs.next()) {
20                  int empID = rs.getInt("ID");
21                  String first = rs.getString("FirstName");
22                  String last = rs.getString("LastName");
23                  Date birthDate = rs.getDate("BirthDate");
24                  float salary = rs.getFloat("Salary");
25                  System.out.println("Employee ID:   " + empID + "\n"
26                  + "Employee Name: " + first + " " + last + "\n"
27                  + "Birth Date:    " + birthDate + "\n"
28                  + "Salary:        " + salary);
29              } // end of while
30          } catch (SQLException e) {
31              System.out.println("SQL Exception: " + e);
32          } // end of try-with-resources
33      }
34  }
```

ExecuteUpdate() - example

```
1. public class InsertJDBCExample {
2.     public static void main(String[] args) {
3.         // Create the "url"
4.         // assume database server is running on the localhost
5.         String url = "jdbc:oracle:thin@localhost:1521:xe";
6.         String unm = "hr";
7.         String pwd = "hr";
8. try {
9. Connection con = DriverManager.getConnection(url, unm, pwd))
10. Statement stmt = con.createStatement();
11. String query = "INSERT INTO Employee VALUES (500, 'Jill',
   'Murray','1950-09-21', 150000)";
12.if (stmt.executeUpdate(query) > 0) {
13.   System.out.println("A new Employee record is added");
14.   }
15.String query1="select *  from Employee";
16.ResultSet rs = stmt.executeUpdate(query1);
17.//code to display the rows
18.}
```

PreparedStatement

- *PreparedStatement* is a subclass of *Statement* that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000.00;
String query = "SELECT * FROM Employee WHERE
  Salary > ?";
PreparedStatement pStmt =
  con.prepareStatement(query);
pStmt.setDouble(1, value);
ResultSet rs = pStmt.executeQuery();
```

- *PreparedStatement* is useful when you want to execute a SQL statement multiple times.

PreparedStatement

Methods

ResultSet executeQuery( )

int executeUpdate( )

boolean execute( )

ResultSet getResultSet ( )

int getUpdateCount( )

void setString(int parameterindex , String x)

void setBoolean(int parameterindex , boolean x)

void setInt(int parameterindex , int x)

void setFloat(int parameterindex , float x)

void setDate(int parameterindex , java.sql.Date x)

void clearParameters( )

PreparedStatement: Setting Parameters

- In general, there is a **`setXXX`** method for each type in the Java programming language.

- **`setXXX`** arguments:
  - The first argument indicates which question mark placeholder is to be set.
  - The second argument indicates the replacement value.
- For example:

```
pStmt.setInt(1, 175);
pStmt.setString(2,"Charles");
```

PreparedStatement:Example

```
PreparedStatement updateEmp;
String updateString = "update Employee“
    + "set SALARY= ? where EMP_NAME like ?";
updateEmp  = con.prepareStatement(updateString);
int[] salary = {1750, 1500, 6000, 1550, 9050};
String[] names = {"David", "Tom", "Nick“,"Harry", "Mark"};
for(int i=0:i<names.length;i++)
 {
     updateEmp.setInt(1, salary[i]);
     updateEmp.setString(2, names[i]);
     updateEmp.executeUpdate();
 }
```

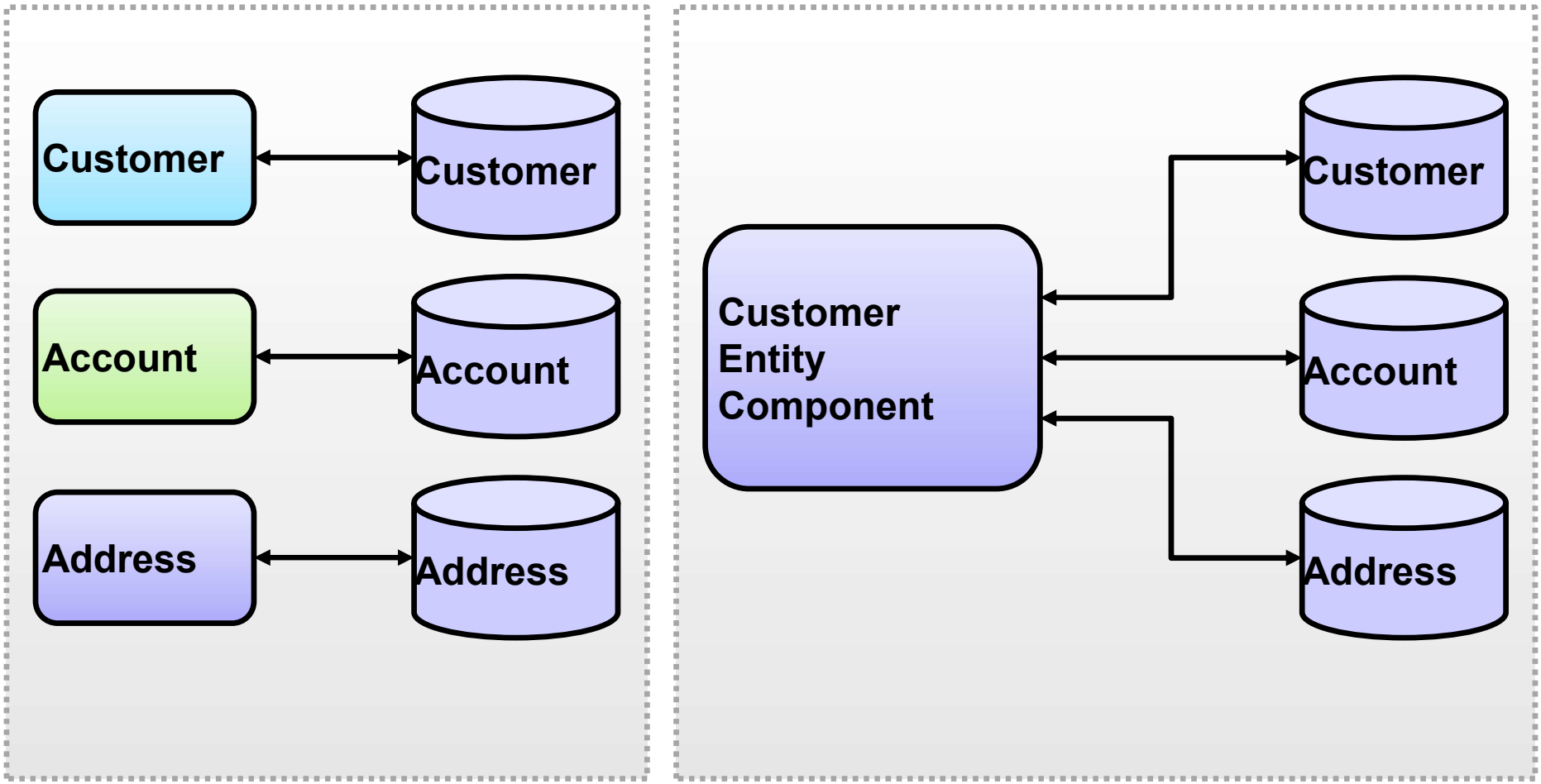
J P A

Java Persistence API: Overview

The Java Persistence API (JPA) is a lightweight framework that leverages Plain Old Java Objects (POJOs) for persisting Java objects that represent relational data (typically in a database).

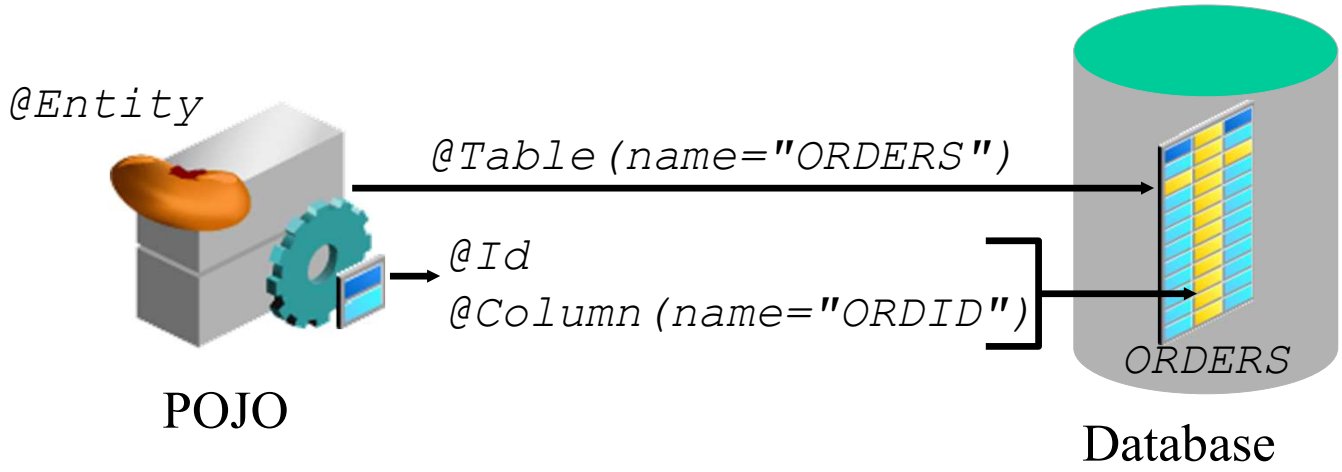
- JPA is built on top of JDBC and addresses the complexity of managing both SQL and Java code.
- JPA is designed to facilitate object-relational mapping.
- JPA works in both Java SE and Java EE environments.
- Key JPA concepts include:
  - Entities
  - Persistence units
  - Persistence contexts

Object-Relational Mapping



What Are JPA Entities?

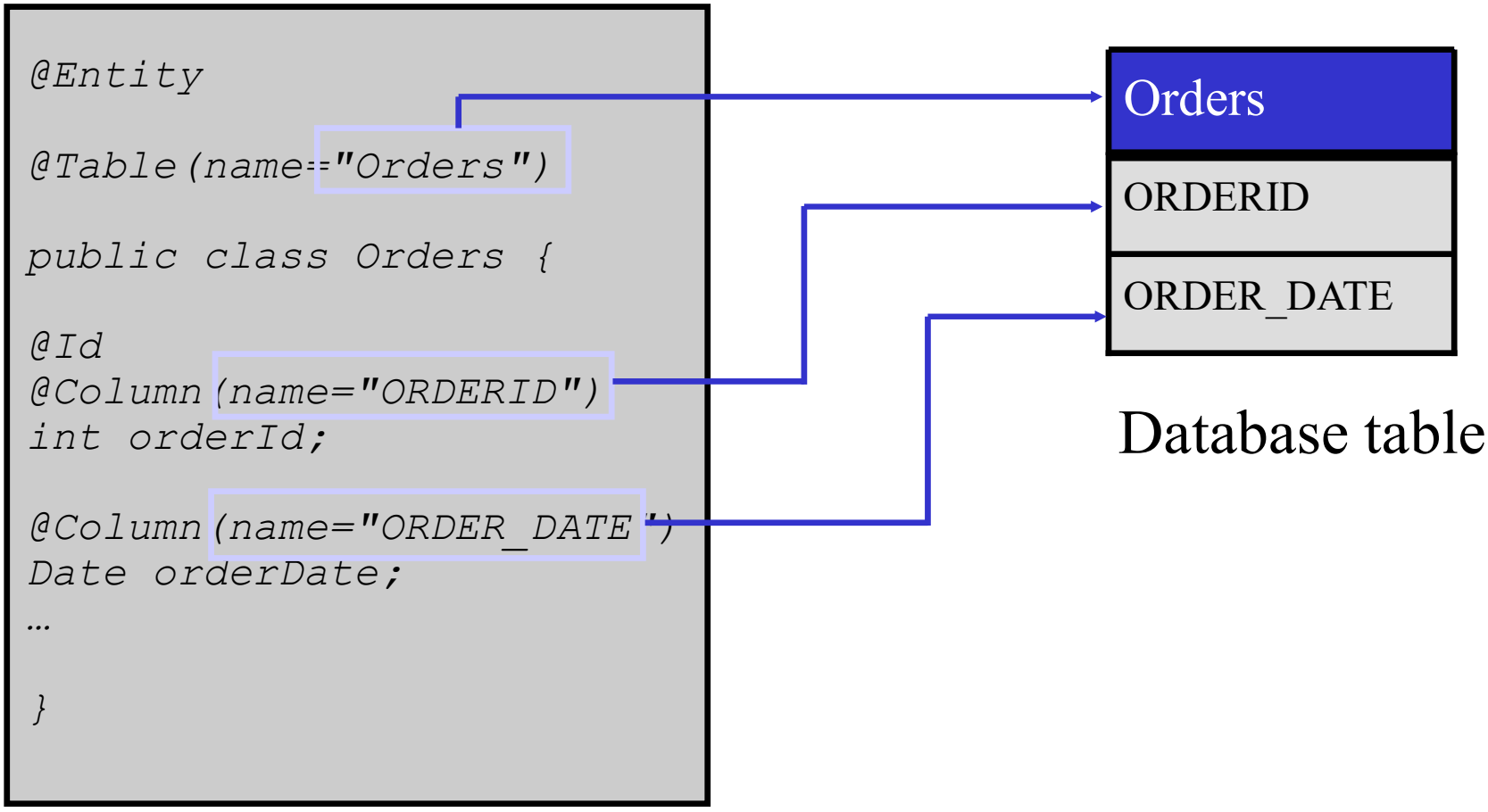
- A Java Persistence API (JPA) entity is:
  - A lightweight object that manages persistent data
  - Defined as a Plain Old Java Object (POJO) marked with the `@Entity` annotation



JPA Entity annotations

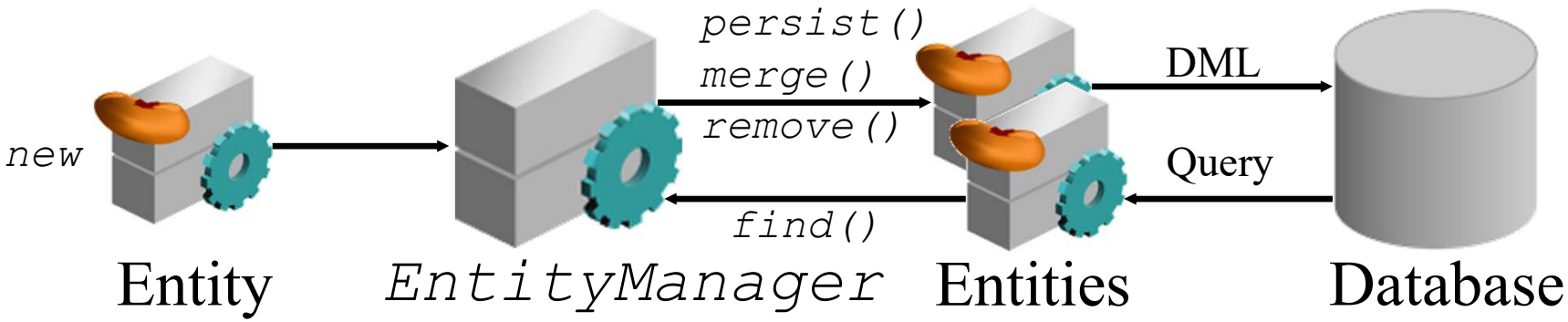
- The following annotations are used to map a POJO to a relational data construct:
  - `@Table` maps the class to a table.
  - `@Column` maps a field to a column (required if the field and column names are different)
  - `@Id` identifies primary key fields

JPA Entity mapping



Managing Persistence of Entities

- The life cycle of an entity is managed by using the *EntityManager* interface, which is part of the JPA.
- An entity can be created by using:
  - The *new* operator (creates detached instance)
  - The *EntityManager* Query API (synchronized with the database)
- An entity is inserted, updated, or deleted from a database through the *EntityManager* API.



EntityManager interface

- The EntityManager interface provides:
  - The find() method to retrieve a database row and instantiate an entity copy
  - Access to a Query API for creating and executing queries based on either of the following:
    - Java Persistence Query Language (JPQL)
    - Native SQL statements
  - Methods to perform persistent operations such as -
    - persist() to mark a new instance for insertion into the database
    - merge() to integrate (either insert or update) an instance into the database
    - remove() to remove an instance from the database

Declaring an Entity

- Declare a new Java class with a no-arg constructor.
- Annotate it with `@Entity`.
- Add fields corresponding to each database column:
  - Add setter and getter methods.
  - Use the `@Id` annotation on the primary key getter method
- If `@Table` annotation is omitted, the class name is mapped to table name
- If `@Column` annotation is omitted, the field names are mapped to the column names

Declaring an Entity - example

```
@Entity
public class Customer implements
    java.io.Serializable {

    @Column (name = "CUSTID")
    private int customerID;
    private String name;

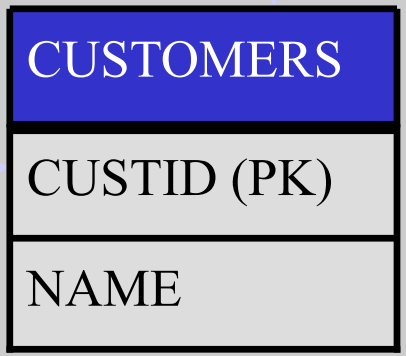
    public Customer() { ... }  // no-arg
    constructor
    @Id                        // annotation
    public int getCustomerID() { ... }
    public void setCustomerID(int id) { ... }
    public String getName() { ... }
    public void setName(String n) { ... }
}
```

Mapping Entities

- Mapping of an entity to a database table is performed:
  - By default
  - Explicitly using annotations or in an XML deployment descriptor

```
@Entity
@Table(name="CUSTOMERS")
public class Customer implements java.io.Serializable {

  @Id
  @Column(name="CUSTID")
  private int customerID;
  private String name;
  …
  public int getCustomerID() { ... }
  public void setCustomerID(int id) { ... }
  public String getName() { ... }
  public void setName(String n) { ... }
}
```



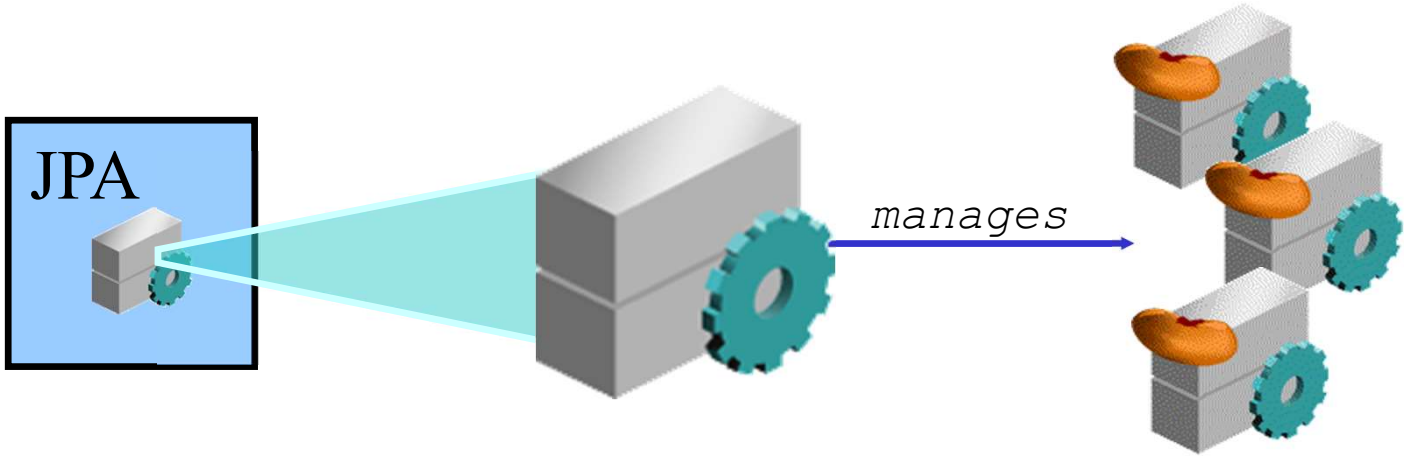
Persistence Unit

- Persistent Unit encapsulates database details, entity class details and persistence provider details.
- Defined in persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"       version="2.0">
    <persistence-unit name="JPA1“  transaction-type="RESOURCE_LOCAL">
       <provider>org.hibernate.ejb.HibernatePersistence</provider>
           <properties>
            <property name="javax.persistence.jdbc.url“ value="jdbc:postgresql://localhost/postgres" />
            <property name="javax.persistence.jdbc.user" value="postgres" />
           <property name="javax.persistence.jdbc.password" value="postgres" />
           <property name="javax.persistence.jdbc.driver"           value="org.postgresql.Driver" />
           <property name="hibernate.hbm2ddl.auto" value="create" />
           </properties>
    </persistence-unit>
</persistence>
```

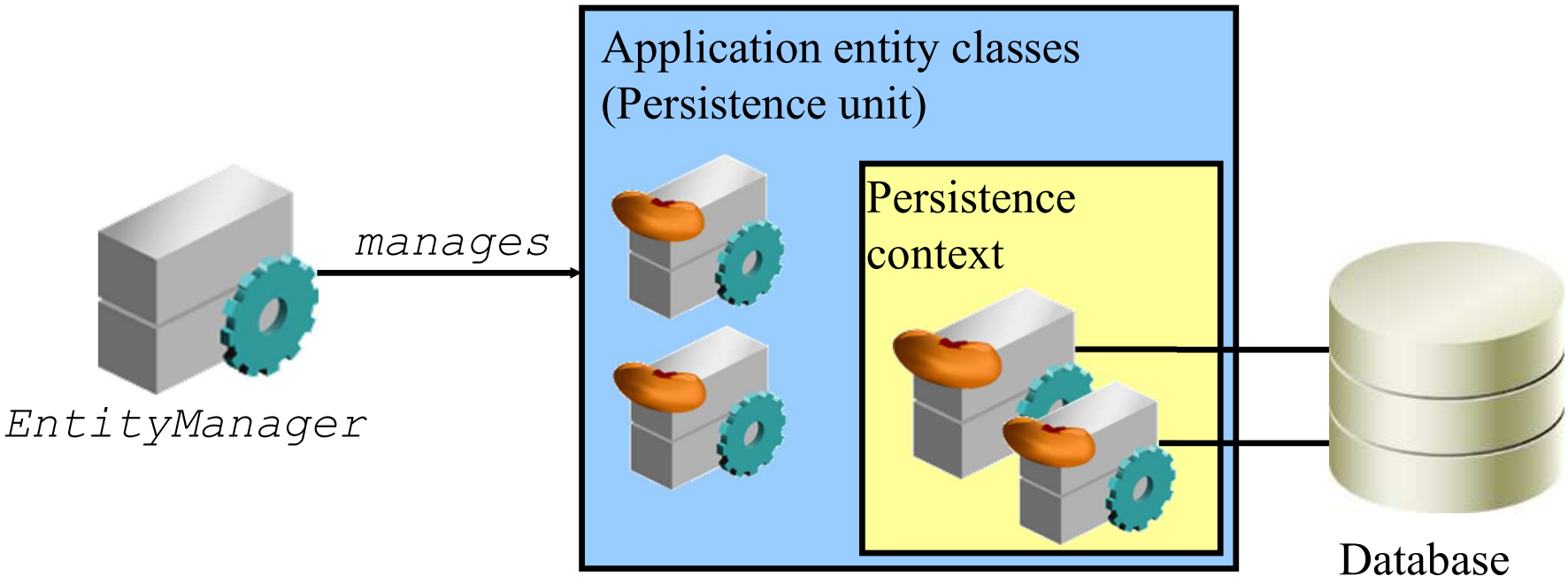
What is *EntityManager*?

- *EntityManager*:
  - Is an interface defined in JPA
  - Is a standard API for performing CRUD operations for entities
  - Acts as a bridge between the object-oriented and the relational models



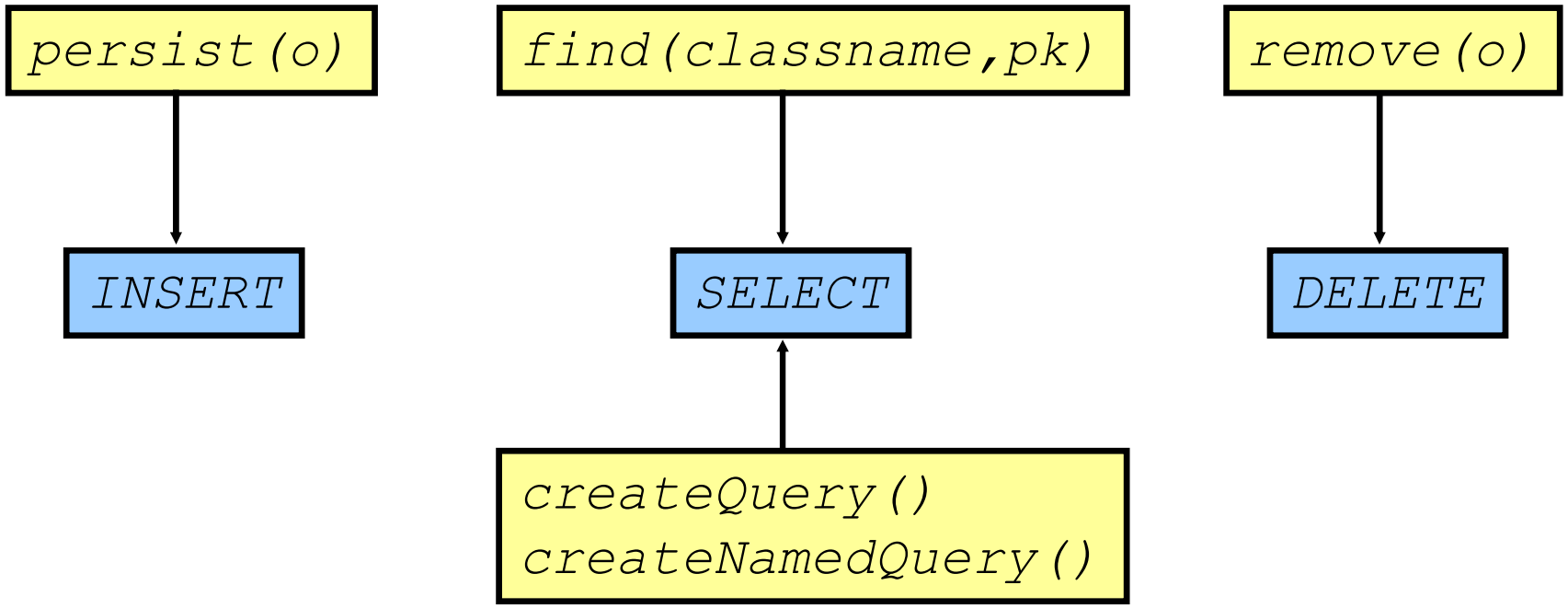
What Is *EntityManager*?

- *EntityManager* is:
  - Associated with a persistence context
  - An object that manages a set of entities defined by a persistence unit



Database Operations with *EntityManager* API

- The *EntityManager* API provides the following methods that map to CRUD database operations:

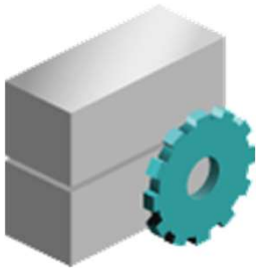


What Is JPA *Query* API?

- The JPA *Query* API:
  - Includes:
    - *EntityManager* methods to create queries
    - *Query* interface methods for executing queries
    - Java Persistence Query Language (JPQL)
  - Supports:
    - Named queries
    - Dynamic queries

Retrieving Entities by Using the *Query* API

- The *EntityManager* interface provides the *Query* API methods to execute JPQL statements:



EntityManager

- *createQuery(String jpql)*
- *createNamedQuery(String name)*

Query instance methods:

- *setParameter(String, Object)*
- *Object getSingleResult()*
- *List getResultList()*
- *Query setMaxResults(int)*
- *Query setFirstResult(int)*
- *int executeUpdate()*

Writing Dynamic Queries

- Example: Find service requests by primary key and a specified status.

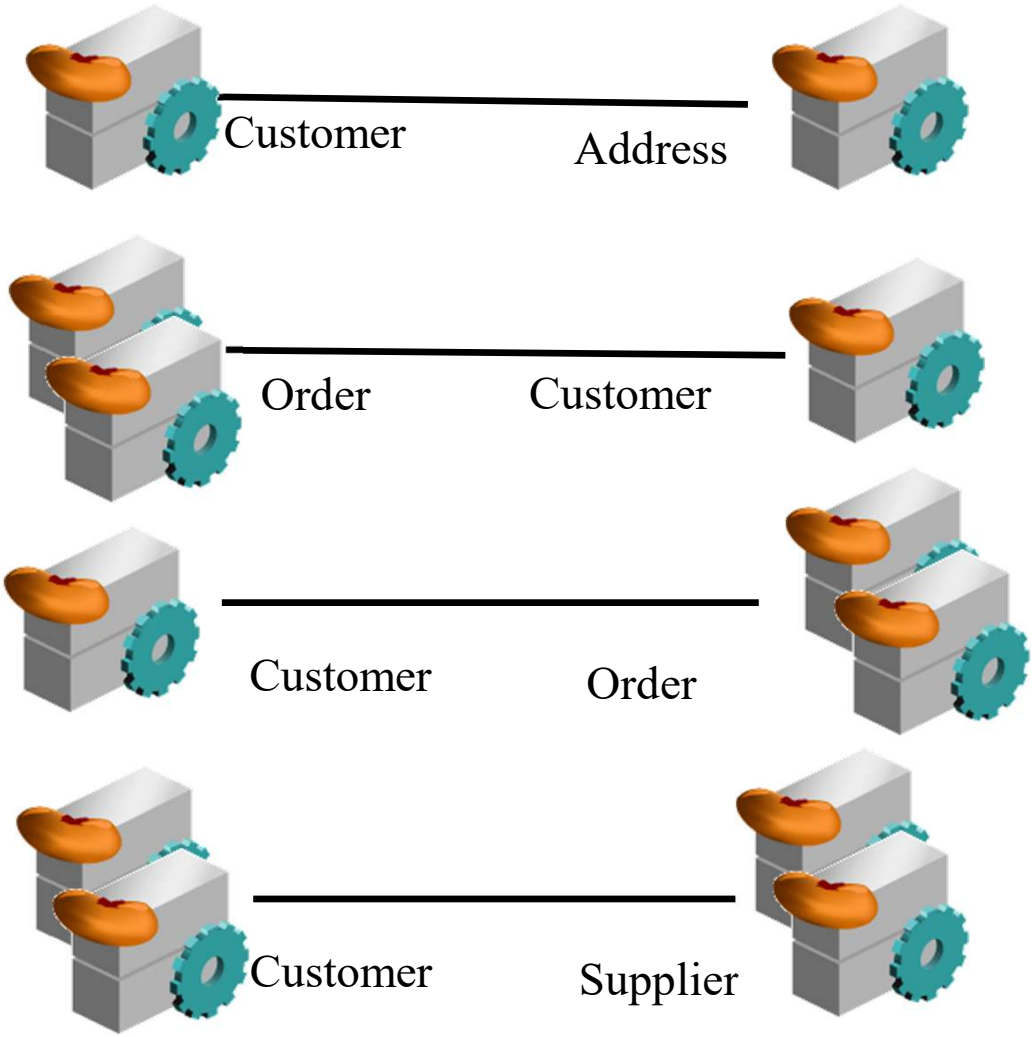
```
Query query = em.createQuery(
 "select sr from ServiceRequests sr " +
 "where sr.svrId = :srvId and sr.status = :status");

query.setParameter("srvId", 100);
query.setParameter("status", 0);

List list = query.getResultList();
```

Mapping Relationships Between Entities

- Annotations for entity relationships:
  - `@OneToOne`
  - `@ManyToOne`
  - `@OneToMany`
  - `@ManyToMany`



Mapping Inheritance

- ***SINGLE_TABLE:***
  - The entities from different classes with a common ancestor are placed in a single table.
  - @DiscriminatorColumn identifies each class
- ***JOINED:***
  - Common base table, with joined subclass tables.
  - Each entity in the hierarchy maps to its own dedicated table that maps only the fields declared on that entity
  - The root entity in the hierarchy is known as the base table, and the tables for all other entities in the hierarchy join with the base table
- ***TABLE_PER_CLASS:***
  - Single-table-per concrete entity class
  - This strategy maps each entity to its own dedicated table

Lambda Expressions

Functional Interface

- Functional Interface is an interface having exactly one abstract method
- Such interfaces are marked with optional @FunctionalInterface annotation

```
@FunctionalInterface
interface xyz {
        //single abstract method
}
```

Lambda Expression

```
public class MaxFinderImpl implements MaxFinder {
        @Override
        public int maximum(int num1, int num2) {
                        return num1>num2?num1:num2;
        }
}
```

```
MaxFinder finder = (num1,num2) -> num1>num2?num1:num2;
int result = finder.maximum(10, 20);
```

Return type of lambda is Functional Interface!

Lambda Expression

- Lambda expression represents an instance of functional interface
- A lambda expression is an anonymous block of code that encapsulates an expression or a block of statements and returns a result
- Syntax of Lambda expression:

```
(argument list) -> { implementation }
```

- The arrow operator -> is used to separate list of parameters and body of lambda expression

Lambda Expression

- Sample Lambda Expressions

Functional Method	Lambda Expression
int fun(int arg);	(num) -> num + 10
int fun(int arg0,int arg1);	(num1, num2) -> num1+num2

int fun(int arg0,int arg1);	(num1, num2) -> {
	int min = num1>num2?num2:num1;
	return min; }

String fun();	() -> “Hello World!”
void fun();	() -> { }
int fun(String arg);	(str) -> str.length()
int fun(String arg);	str -> str.length()

Built-in Functional Interfaces

- Java SE 8 provides a rich set of 43 functional interfaces
- All these interfaces are included under package java.util.function
- This set of interfaces can be utilized to implement lambda expressions
- All functional interfaces are categorized into four types:
  - Supplier
  - Consumer
  - Predicate
  - Function

Supplier

- A Supplier<T> represents a function that takes no argument and returns a result of type T.
- This is an interface that doesn't takes any object but provides a new one

```
@FunctionalInterface
public interface Supplier<T> {
        T get();
}
```

- List of predefined Suppliers:
  - BooleanSupplier
  - IntSupplier
  - LongSupplier
  - DoubleSupplier etc.

Consumer

- A Consumer<T> represents a function that takes an argument and returns no result
- A BiConsumer<T,U> takes two objects which can be of different type and returns nothing

```
@FunctionalInterface
public interface Consumer<T> {
        void accept(T t);
}
```

```
@FunctionalInterface
public interface BiConsumer<T,U> {
        void accept(T t, U,u);
}
```

- List of predefined Consumer:
  - IntConsumer
  - LongConsumer etc.

Predicate

- A Predicate<T> represents a function that takes an argument and returns true or false result
- A BiPredicate<T,U> takes two objects which can be of different type and returns result as either true or false

```
@FunctionalInterface
public interface Predicate<T> {
        boolean test(T t);
}
```

```
@FunctionalInterface
public interface BiPredicate<T,U> {
        boolean test(T t, U,u);
}
```

- List of predefined Predicates:
  - IntPredicate
  - LongPredicate
  - DoublePredicate etc.

Function

- A Function<T> represents a function that takes an argument and returns another object
- A BiFunction<T,U> takes two objects which can be of different type and returns one object

```
@FunctionalInterface
public interface Function<T,R> {
        R apply(T t);
}
```

```
@FunctionalInterface
public interface BiFunction<T,U,R> {
        R apply(T t, U,u);
}
```

Lambda Expressions for Function Interfaces

Functional Interface	Functional Method	Lambda Expression
Supplier<String>	String get();	() -> “Hello World”;
BooleanSupplier	boolean get();	() -> { return true; }
Consumer<String>	void accept(String str);	(msg) -> System.out.println(msg);
IntConsumer	void accept(int num);	(num) -> System.out.println(num);
Predicate<Integer>	boolean test(Integer num);	(num) -> num>0;
Function<String,Integer>	Integer apply(String str);	(str) -> str.length();
UnaryOperator<Integer>	Integer apply(Integer num);	(num) -> num +10;
BiFunction<String,String, Boolean>	Boolean apply(String user,String pass);	(user,pass) -> { //functionality to validate user }

Using Built-in Functional Interfaces

```
Consumer<String> consumer = (String str)-> System.out.println(str);
consumer.accept("Hello LE!");
Supplier<String> supplier = () -> "Hello from Supplier!";
consumer.accept(supplier.get());
//even number test
Predicate<Integer> predicate = num -> num%2==0;
System.out.println(predicate.test(24));
System.out.println(predicate.test(20));
//max test
BiFunction<Integer, Integer, Integer> maxFunction = (x,y)->x>y?x:y;
System.out.println(maxFunction.apply(25, 14));
```

Stream API

Group of Employees (Collections)

How to find the most senior employee?

What is the count of employees joined this year?

Send meeting Invite to only Java Programmers

Manager

Stream API

- Stream API allows developers process data in a declarative way.
- Enhances the usability of Java Collection types, making it easy to iterate and perform tasks against each element in the collection
- Supports sequential and parallel operations

Stream Operations

- Stream defines many operations, which can be grouped in two categories
  - Intermediate operations
  - Terminal Operations
- Stream operations that can be connected are called intermediate operations.

  They can be connected together because their return type is a Stream.
- Operations that close a stream pipeline are called terminal operations.

Intermediate operations

Terminal operation

Working with Stream:

- A stream pipeline consist of source, zero or more intermediate operations and a terminal operation
- A stream pipeline can be viewed as a query on the stream source
- Operations on stream are categories as:
  - Filter
  - Map
  - Search
  - Sort

Stream Interface

- The Stream API consists of the types in the java.util.stream package
- The “Stream” interface is the most frequently used stream type
- A Stream can be used to transfer any type of objects
- Few important method of Stream Interface are:

concat	count
collect	filter
forEach	limit
map	max
min	sum
reduce	sorted

Intermediate
Terminal

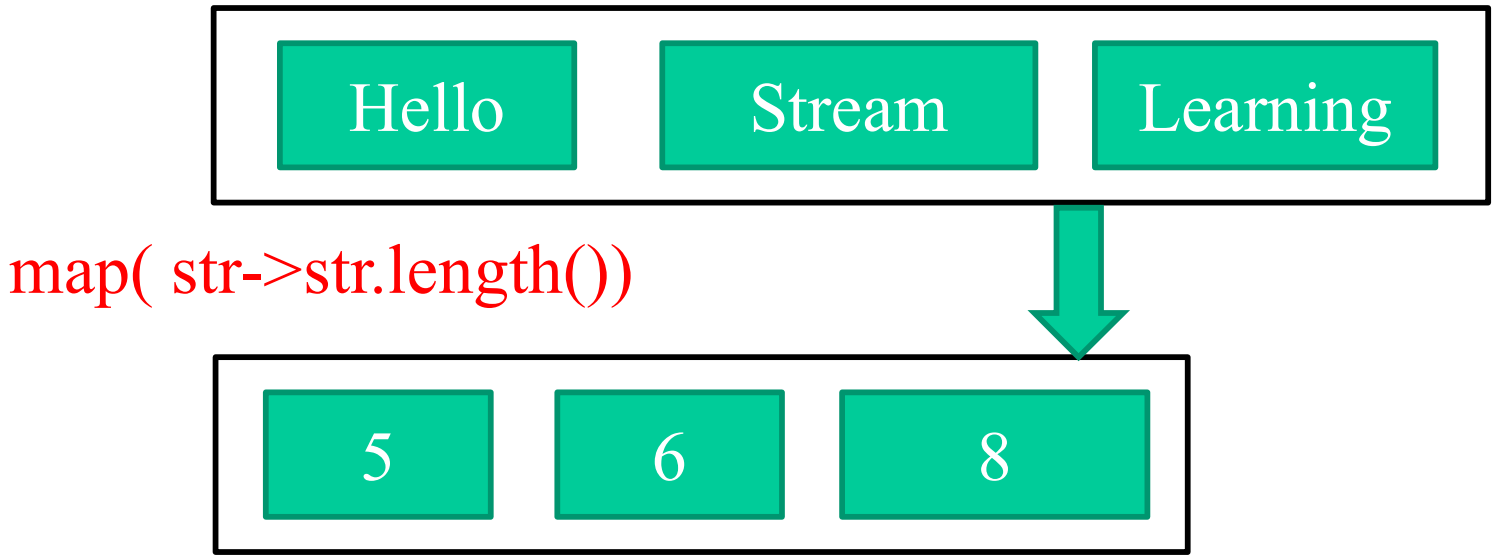
Mapping

- The map method maps each element of stream with the result of passing the element to a function.
- map() takes a function (java.util.function.Function) as an argument to project the elements of a stream into another form.
- The function is applied to each element, “mapping” it into a new element.
- The map method returns a new Stream of elements whose type may be different from the type of the elements of the current stream.

Mapping Example

```
List<String> words = Arrays.asList("Hello","Stream","Learning");

words.stream()
   .map(str->str.length())
   .forEach(System.out : : println);
```



Filtering

- There are several operations that can be used to filter elements from a stream:

Operation	What ?
filter(Predicate)	Takes a predicate (java.util.function.Predicate) as an argument and returns a stream including all elements that match the given predicate
distinct	Returns a stream with unique elements (according to the implementation of equals for a stream element)
limit(n)	Returns a stream that is no longer than the given size n
skip(n)	Returns a stream with the first n number of elements discarded

Filtering Examples

- filter(predicate)

```
List<Integer>  listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().filter(num -> num > 10).forEach(num->System.out.println(num));
```

11 44 66 33 44

- distinct()

```
List<Integer>  listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().distinct().forEach(System.out : : println);
```

11 3 44 5 66 33

- limit(size)

```
List<Integer>  listInt = Arrays.asList(11,3,44,5,66,33,44);
listInt.stream().limit(4).forEach(System.out : : println);
```

11 3 44 5

Sorting

```
List<String> words = Arrays.asList(“Hello”,”Stream”,”Learning”);

words.stream()
   .sorted()
   .forEach(System.out : : println);
```